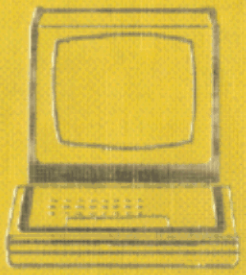
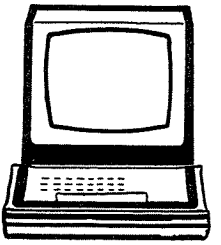
THE A TO Z BOOK OF COMPUTER GAMES



BY THOMAS C. McINTIRE

No. 1062
$12.95

THE A TO Z BOOK OF COMPUTER GAMES



BY THOMAS C. McINTIRE

TAB BOOKS Inc.
BLUE RIDGE SUMMIT, PA. 17214

FIRST EDITION

SEVENTH PRINTING

Printed in the United States of America

Reproduction or publication of the content in any manner, without express permission of the publisher, is prohibited. No liability is assumed with respect to the use of the information herein.

Copyright © 1979 by TAB BOOKS Inc.

Library of Congress Cataloging in Publication Data

McIntire, Thomas C. 1942-
The A to Z book of computer games.

Includes index.
1. Electronic digital computers—Programming. 2. Basic (Computer program language) 3. Games—Date processing. I. Title.
QA76.6M322 794 79-11698
ISBN 0-8306-9809-4
ISBN 0-8306-1062-6 pbk.

Cover photo courtesy of IBM General Systems Division.

Contents

Introduction 7
Programming Notes—Program Format—Program Names

A **Abstract** 15
The Programming Problem—The Design Approach—Building the Program—How the Programs Works—The Program

B **Bandit** 26
Defining the Problem—Design Strategy—The Internals of Bandit—The Program

C **Cokes** 34
How the Program Plays—The Program

D **Dice** 38
Problem Definition—The Architecture of Dice—A General-Purpose Technique—How Dice Works—The Program

E **Elevate** 47
How the Program Works—The Program

F **Fivecard** 54
Fivecard Design—How Fivecard Works—Notes About the Architecture—How the Mainline Works—Fivecard Hand Analyzer—Fivecard Winner Picker—The Program

G **Gunners** 76
Building Gunners—Gunners' Internals—The Program

H **Hotshot** 87
the Virtual Logical Grid—The Programmed Strategy—Putting Hotshot Together—Making Hotshot Work—The Program

I **Invert** 101
Program Overview—What's Inside of Invert—The Program

J **Justluck** 115
Justluck's Architecture—Justluck's Listing—The Program

K **Knights** 125
The Layout of Knights—Knights From Within—The Program

L **Lapides** 138
Lapides Logic—The Program

M **Match** 144
Match From the Outside—Match From the Inside—The Program

N **Naughts & Crosses** 156
A Design Approach—The Tables in Naughts & Crosses—Modules, Mapping & More—Housekeeping & Mainline—The Program

O **O-Tell-O** 172
The Play—Problem Definition—The Basic Logic Principles—A Picture of the Program—O-tell-O From the Inside—The Program

P **Par-2** 191
Design Considerations—Programming Par-2—The Program

Q **Quantal** 199
Design & Logic—The Program

R **Roulette** 206
Playing Roulette—Programming Roulette—The Program

S **States** 217
The States Template—States Internally—The Program

T **Twenty 1** 228
The Deck—A Design Audit—Putting Twenty1 Together—The Program

U **Ultranim** 240
Strategies in Nim—The Computer's Choice—The Rest of Ultranim—The Program

V **Verboten** 252
A Verboten Program—Verboten's Coding—The Program

W **Wampus** 260
Defining the Problem—Describing the Program—Describing the Mechanics—The Program

X **Xchange** 270
The Game—Program Organization—Print Grids—The End—The Program

Y **Yat-C** 282
The Program Template—The Coding of Yat-C—The Program

Z **Z-End** 294
From The Top—Replay—The Program

Appendices

1 **BASIC Dialects: Foreign Conversions** 299

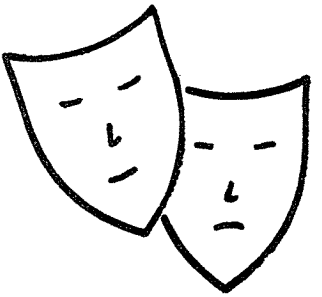
2 **RANDOMIZE Methods** 302

3 **Game Matrix** 304

4 **Truth Table Blanks: 8 Bits by 256 Words** 305

Index 307

Introduction



There is probably no need to explain the fun you can have playing games with computers. More specious though, perhaps, is the enjoyment experienced by the person who programs the computer. Most games played with cards, dice, and the like are challenging and mentally stimulating. So too, is the process of computer programming—perhaps even more so.

Nearly all games allow for some element of luck, but computer programming is more finite; in fact lucky breaks are not really a factor of carefully designed programs. Yet the quickening of the pulse in anticipation of the next draw of cards is not very different from that experienced by a programmer when he runs his newly written program the first time.

There is another comparison of games and programming. Matching wits with a skillful opponent is somewhat akin to mastering a machine. Perservering, finally getting an intricate program to operate correctly is much the same as succeeding in a bout with a clever adversary. And after a match of either type, as the warm feeling of victory begins to wane, the thrill of the contest can be rekindled with thoughts about future challenges.

Just as in games of skill, programming requires patience, practice, and diligence to maintain high levels of proficiency. The optimum combination of these two challenges— games and programming — is to devise a program that enables the computer to play a superior game, difficult even for its programmer to beat.

Computers are useful appliances in games. Advances in micro-electronics in recent years have reduced the cost of

computer-based devices to the point where many machines are available for purely entertainment purposes. Coming rapidly into vogue now are devices for which a variety of game programs may be purchased.

These exploit attributes such as the speed, arithmetic, precision, or the impartiality of computers. The machine may serve as the game board, playing field, or arena for human contestants. Scorekeeping and refereeing are often features: it's tough to argue convincingly with a machine. The more sophisticated computer driven games may also have a player role, allowing for human-machine contests.

This book is not offered in competition to these technical marvels of the seventies, nor with the many books that offer upwards of a few hundred game programs ready to be copied and played. Of course the programs contained here may simply be copied. Certainly they are fun. These selections intend a dual purpose, however.

Do copy and play these games, but study them also as computer programs. The variety from *A* to *Z* was carefully picked to provide examples of gaming schemes appropriate to computers as well as tricks and techniques for programming in BASIC.

The narrative of each chapter explains the scheme of the game, of course, but provided also are lessons on game program construction. I have also tried to insure that there is at least one programming trick-of-the-trade per listing, useful for other types of programs and for languages other than just BASIC. The purpose is to equip the student for the maximum of enjoyment in playing games and programming computers.

Some assumptions were made in choosing the games that follow. BASIC, beginner's all-purpose symbolic instruction code, is a common programming language. In the several years since BASIC was invented, however, numerous variations have appeared. Insofar as possible there was an honest attempt to provide programming examples that are not particularly sensitive to a specific language implementation.

The burgeoning microcomputer industry has adopted BASIC almost universally. Various factions have taken liberties with the original syntax, and each vendor's product has nuances— even among those claiming they conform to the recent standard from the American National Standards Institute. The language, conceived at Dartmouth, made it easy to program a computer from a terminal on a time-sharing system. The inventors could not anticipate the popu-

larity BASIC would eventually attain, though, nor the permutations of its form that would be forced by today's technology.

Many BASIC dialects are a direct outcome of its use in desktop microcomputers, and extensions to the original language have evolved to support modern peripherals. These influences were factored also in designing the programs for this book.

The CRT (cathode-ray tube), the TV-set-like display, is especially conducive to computerized games. All of the programs in this book were developed on a CRT-based machine, but they will work well enough on one with a serial printer. To facilitate easy implementation on either, and to minimize device-specific language features, all output formatting has been kept simplistic. As an aid to those wishing to copy these games just as they are presented, Appendix I deals with the subject of BASIC language conversions. Many of the more common differences in dialects are identified there.

Simplicity was attempted also in the internal design of these programs. The practiced BASIC programmer, versed in a favorite product, will find numerous opportunities to optimize these designs. And to the professional offering critique, be aware that I believe in the KISS* philosophy, especially for the benefit of our novitiates.

In all events, it is my hope that you will be able to quickly perceive the nucleus of each of these games and that you will enjoy full license to enhance and otherwise modify them to your own liking. I hasten to add, however, that this invitation is intended for personal endeavors and not for commercial plagiarism.

There is no special significance to the order of presentation of the chapters that follow, other than the obvious one of using the alphabet, keying on the first letter of the program name. The shortest is Cokes, perhaps the simplest is Z-End (a variation of the classic *nim*), and unquestionably the longest program is Fivecard (poker). A cross-reference appendix at the back of the book will help you to locate different game types, programming techniques, algorithms, etc.

Do read the notes that follow before immersing yourself in any of these programs. It is here that I have described common aspects of the structure of most of the programs and offered other comments of a general nature to preclude repetition in each chapter.

PROGRAMMING NOTES

Chance, the flip of a coin or a random draw at cards, is an element of many games. To expect a computer to supply a

*Keep it simple, stupid.

haphazard value is nearly contrary to the principles on which computers are designed, however. Precision is an inherent feature of computers and normally nothing occurs by chance. Random-number generation through program routines is the usual solution to this seeming paradox.

To the serious student of mathematics the topic of random numbers offers several tangents for study. We shall accede to the many texts that treat this subject professionally. Definition here is only for that which has been provided, why, and how to use it.

The basic problem is to have the computer hand up a number that cannot be presaged by a human player. Ideally, even the programmer should not be able to predict exactly what number will come forth. To obtain a random value in BASIC usually a language function RND is used. The model used in the development of these programs requires a numeric expression with the function, such as: RND(1).

Any positive integer used will cause the system to return a six-digit decimal value ranging between zero and one (example: □ .372508). If the function is written with a zero expression, such as RND(0), the value returned will be the same as that the last previous fetch brought forth.

Many implementations of BASIC use the key word RANDOMIZE to start the system's random-number generator from a system-supplied "seed." Some, however, expect the programmer to supply the root number from which subsequent calls can derive a pseudorandom value. Appendix II anticipates this, and contained there is a BASIC programming example for using innocuous operator input to start the ball rolling.

All of the programs listed here that have need for random numbers contain a *GOSUB 9000* early in the sequence. The subroutine listed at statement 9000 is simply the RND(1) followed by a RETURN. You may either use what we have here or pick up the subroutine from the appendix. Of course, if your BASIC permits RANDOMIZE, simply substitute it for the subroutine jump instruction and ignore the statements after line 9000.

A note is appropriate also on the use of alphabetic responses from players. Letters *Y* and *N* for *yes* and *no* are typically used. Our model allows for alphanumeric string input. For versions of BASIC without this facility, or if your preference differs, numeric input conventions may be substituted. For example, you may use 1 for yes and 0 for no, or some similar code.

Other conventions of design prevailed but were not always practiced religiously. Symbol Q (for query) was most often used for

answers solicited from the operator, and the letters *I* and *J* were used extensively for the counters in FOR-NEXT loops.

We come now to the subject of program structure or, in the argot of the trade, *software architecture.* System or program housekeeping tasks are up front, just before the operator is asked whether the rules or instructions should be printed. For most programs the rules are listed starting at line 1000 and end with a subroutine RETURN. Housekeeping for the game itself is usually next, and all of that immediately preceding the start of the game is never again accessed.

Most game processes can be grouped by major function, such as create a deck of cards in memory (in a table), shuffle the deck using a random-number device and draw the cards from the deck one at a time. Typically then, each such function is coded as a stand-alone subroutine. One advantage is that the separate subroutines may be accessed whenever there is a need, such as to reshuffle the deck in blackjack (name, *Twenty1).*

Building programs in this manner—working on one module at a time—is often easier, especially in an interactive form of BASIC. During development of a program, I usually code (and load) one module at a time. Debugging is easier this way, too, since new parts are not added until all that has been entered before is working properly.

A mainline routine serves to bind all of the subroutines together into a complete program. A start sequence will establish score counters at zero and set up the playing order. A series of subroutine jumps (GOSUB instructions) will call the functional modules as necessary. At the bottom of one complete round of play, the end-of-game condition is tested for, and if the game is not over a GOTO statement reenters the mainline near the top.

If a fallthrough occurs when the end-of-game state is tested for the players are asked whether they would like to play another round. If the answer is yes another GOTO branches to the game's housekeeping area. Otherwise an END statement is encountered and the program terminates.

A caution is in order at this point about switching players and scorekeeping. No particular convention prevails. Scores may be tallied within the mainline or, likely as not, in one of the servicing modules. This is true of the player-switching logic also. A flip-flop scheme is usually used for two-player games, often within the mainline near the top. In multiple-player games (three or more), a player counter is more likely to be updated near the bottom of the

mainline to insure that all players have had their share of turns before the game ends.

Now, a word on documentation. By many standards the use of remarks (REM statements) in the listings is notoriously sparse. It is supposed, however, the narrative and diagrams in each chapter sufficiently explain program logic and structure. A favorite type of diagram used here is the template. I have borrowed the template concept from software engineering labs because of its excellent "big picture" capability. To use the more traditional flowcharts takes more space, and often the overall structure is not as apparent as it is with the template.

Space conservation was a factor throughout. The scarcity of REM statements was in part to save memory and partly to reduce the length of the listings. A tradeoff was made to permit short PRINT lines for those with 32- or 40-column printers or display lines; but sometimes more statements resulted than would have otherwise.

It is difficult to predict the amount of memory required for unidentified systems. As a guide, though, the model used was an 8080-based microcomputer. The BASIC implementation was of the source-code interpretive type, and the amount of user memory available was approximately 4000 bytes. As a SAWG* then, these programs should fit in a 4-kilobyte environment on most systems.

PROGRAM FORMAT

Statement numbering is intended to be always in increments of ten. Very occasionally, the low-order digit sequence will be of the form two, four, six, eight — which will usually mean that some steps were inadvertently left out when the program was first written, but added later. At other times, this stilted sequence was adopted to enable coding of enough steps within an artificial numbering boundary.

Whenever a single-statement insertion was required, perhaps to eliminate a bug, the low-order digit used was five. Whichever the case, the prevalence of multiples of ten is presumed to be sufficient to permit easy modification anywhere without the need to renumber any line referencing instructions.

For most programs two-digit line numbers (00-90) will contain only program startup instructions, executed but once, and not subject to being branched to from within the program anywhere else. Program mainline tasks are usually restricted to three-digit

*SWAG scientifically weighted, arbitrary guess

line numbers. Branching within the mainline area is frequent enough, but as a formatting and design convention there should not normally be any reference to this area by any statement outside of the three-digit range of numbers.

Where the major-task module form of architecture was adhered to, the subroutines are blocked by thousands, that is, line numbers starting with 1000, 2000, 3000, etc. Nested subroutines, subroutines within subroutines, will begin with a whole-hundreds sequence within the thousands block, such as 2400, 2600, 2800, etc.

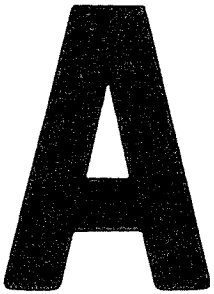
PROGRAM NAMES

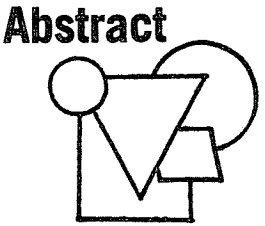
There are never more than eight characters in a program name, and they're always alphanumeric; the only special character used is the hyphen. The model used for developing these programs could only tolerate an eight-character program ID, and I have found this to be something of a de facto standard on many microcomputer products. And my experience on various systems has caused me to adopt a no-spaces, no-punctuation, always-alpha-first set of rules in coining file names. So be it.

Enough of the general; it's time for the games. In the chapters that follow, the game is described first, followed by a brief on how I approached the design, and finally, a description of how the program works.

And *work* they all do. Having programmed for many years, I realize the folly of promising error-free code. We have tried our best to insure that there are no bugs, but in the unlikely event that you do find one, feel perfectly free to exterminate it.

Tom McIntire





Perhaps you know someone who has indicated that he or she would like to learn programming. In a limited way the game of *Abstract* gauges a player's aptitude for programming because it tests for deductive reasoning ability. It is just a game, and an amusing one to play; so don't jeopardize friendships or career aspirations by reason of anyone's consistent inability to win.

The game begins with an announcement by the computer that it is holding a three-digit number in memory. The player has twenty chances to guess what the number is. The first guess must necessarily be just that, an arbitrary guess. Following each entry a list of clues are printed to show the relative accuracy of each of the digits in the player's guess.

The clues that are possible, and their order of presentation is, ASTUTE, ABSTRACT, and ASKEW. For each of the player's digits that corresponds exactly with that of the number being held in memory, the clue ASTUTE will print. Obviously, three astute is a winner.

Each of the digits of the player's guess is compared, one at a time, with each digit of the computer's number. For each of the player's digits that has no match anywhere within the computer's number, the clue that is printed is ABSTRACT.

Whenever a player's digit matches one of the computer's, but the digit is in the wrong position, ASKEW indicates the misalignment. What is not revealed, however, is which of the digits prompts the different clues, since the clues are always presented in the same sequence.

The three types of comparisons used to trigger the clues following each entry are mutually exclusive. By deductive reasoning, then, and with an ounce or two of luck, the player may home in on the correct number. To do so merely requires that each guess be carefully formulated based on the trail of clues provided with previous entries.

A few trial games might be necessary to fully grasp the significance of the clues. Interestingly, this tends to be less true with children sometimes than for some adults.

It is important to be aware that no two digits of a guess should ever be the same, and that the player should not enter a number having more than three digits. Either type of error will invoke an appropriate message, but the guess is not counted as a turn. The game ends when the player's guess matches the computer's number exactly, and the message is: YOU GOT IT. Also, if the total number of guesses is less than thirteen the number of turns used will be printed. If thirteen or more turns are taken, the ending message includes the sarcasm; FINALLY!!!

In either event, or after twenty attempts, the game ends with an option to play again. For those unfortunates who use up all twenty of the permitted guesses without solving *ABSTRACT*, the ending routine will reveal the elusive number that the computer has been holding.

THE PROGRAMMING PROBLEM

Conceptually this is not a diifficult program to design, but it has some interesting processing requirements. Seemingly, two three-digit numbers must be managed with continuing comparisons made of their respective digits. The first number, of random origin, is that held in memory by the computer. The second number is the player's guess; naturally, it's subject to change with each succeeding turn.

In both cases the two numbers must be quality checked to insure that neither contains more than three digits and that neither contains any duplicate digits. Each digit, then, of each of the two numbers must be cross-compared to determine which clues are appropriate.

Housekeeping and maintenance chores are nominal, requiring only that the turns be counted and limited to twenty. Allowing for repeated play without having to reload the program *does* require that the *turn* counter be reset to one at the beginning of each game.

THE DESIGN APPROACH

Since the program will have to examine individual digits repeatedly, set up the memory number as three distinct integers from

the outset. The player's guess is a whole number, however, so a parsing routine is required to separate the digits to facilitate the individual comparisons.

Three different clues means that at least three different tests must be made, so the obvious choice is to use separate routines—one for each type of clue. Because multiple clues can occur for each of the tests, but only a single print line is wanted, the *clue* routines must each provide a counter for use at message-print time.

BUILDING THE PROGRAM

Begin by roughing out the program template shown in Fig. A-1. (The line numbers are added during the writing of the program.) Each of the major tasks and their approximate relationships are

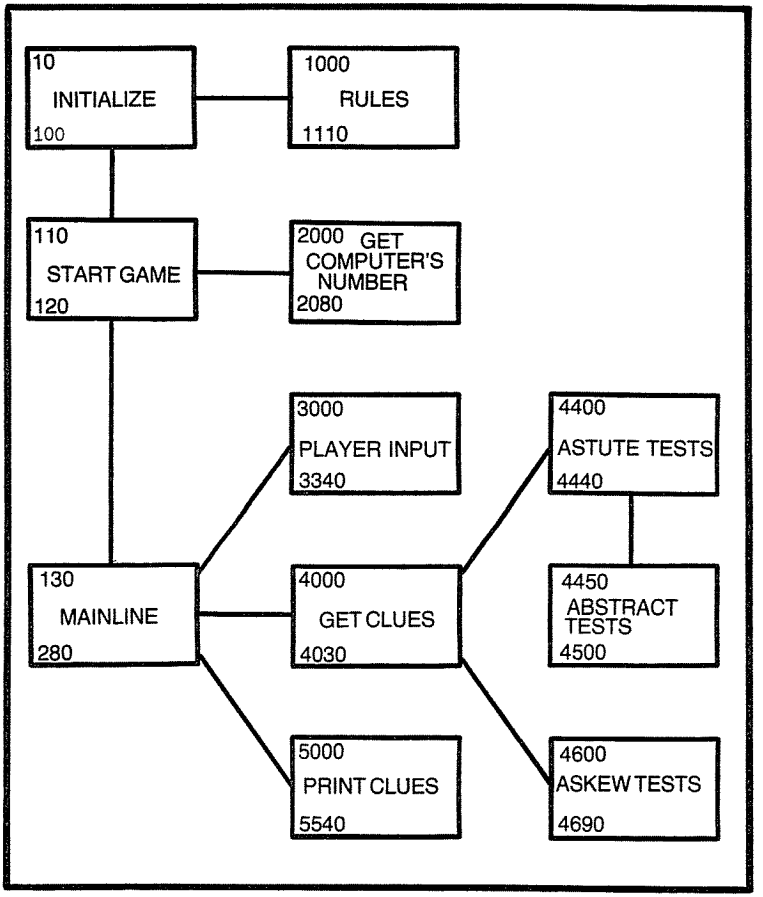


Fig. A-1. Program template for Abstract.

identified there. But the sequence of processing is not yet clearly defined. Since the template does show which routines will ultimately be needed, and by planning for each to be a stand-alone subroutine, the modules can be coded in any order, and depend on the mainline to prescribe the calling sequence.

A suggested starting point is to lay up the number-parser first, since so much depends on being able to correctly split apart the player's guess. Figure A-2 shows the BASIC expressions that can accomplish this and the rudiments of the algorithm on which the parsing scheme is based. The principle alluded to in this example is useful for any processing problem that requires isolation of the individual digits of a whole number.

It's a good idea to code the computer's random-number maker next. This will tend to insure that the symbol letters used for storing each of the player's digits will align correctly with those of the computer. Although this subroutine should be coded at this point a good debugging trick is to wait until later to bind it into the mainline program.

To take full advantage of this idea during early trial runs of the program with the bulk of the programming statements in place, use programmer supplied test-case integers rather than allow the system to generate an unknown value. This scheme will also enable you to qualify the correctness of each of the *clue* generators because it permits easy changes to both the computer's supposedly hidden number and a complete variety of guesses.

Back to the business of building. The three clue testors should be coded next. There is no special importance as to which should be programmed first, second, or third; but within each, extreme care is necessary to insure accuracy — especially in the use of the storage location identifiers for the various parts of the numbers.

The message-print subroutine that outputs the clues is the final major module to be coded. At this point, assuming that each of the clue testers are able to generate a number between one and three (inclusive), all that's needed is a loop-controlled print task that can be accessed three times—to be called once for each of the three types of clues.

Finally the frame is built by which all of the subroutines are managed. Basically the calling sequence is (1) generate the computer's number, (2) accept player input and qualify it for range and duplicates, (3) generate and print the clues, and (4) branch back to permit another guess. Just before the branch at the bottom of the mainline, don't forget to check for a winner (a three in the *astute* counter), nor for the possibility that the *guess* counter has reached the twenty-tries limit.

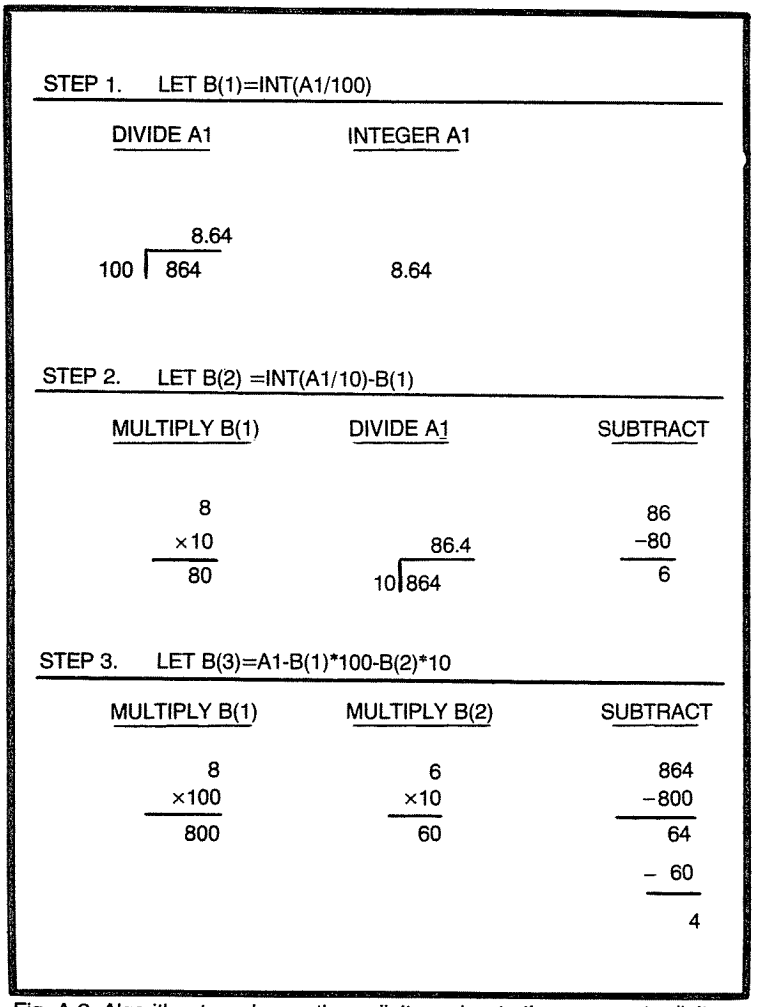


Fig. A-2. Algorithm to reduce a three-digit number to three separate digits.

HOW THE PROGRAM WORKS

Advice to the system to allocate two three-element tables (identified as *A* and *B*) is contained in statement 40. *Table A* will house each of the three digits of the held number, *Table B* is for storing the player's guess after it is parsed each time. It's not difficult to remember which table has which number because the computer's number must be generated first, and *A* comes before *B*.

The logic for permitting an optional printing of the game's rules is easily discernible by studying lines 50 through 90. Notice that the rules themselves are printed by the subroutine starting at line 1000.

A bit of subtlety exits at line 100 as to the logical use of this statement, but the intent is to provide a touch of humor. The alpha-string storage location denoted by F$ is initialized with the phrase "I FORGOT TO TELL YOU THAT". Later, during playing of the game, *in the first instance only*, if the operator enters a three-digit number containing duplicate digits, the contents of F$ are printed, followed by: "MY NUMBER HAS NO 2 DIGITS THE SAME."

The printing sequence starting at line 3100 outputs the "forgotten" instruction, and F$ is immediately reloaded with the phrase "PAY ATTENTION". From then on any duplicate-digits error detected by the sequence starting at line 3060 will cause the operator to be admonished for forgetting the "forgotten" rule.

Housekeeping for the game is done by statements 110 and 120. The symbol G, for guess, is used to store the number of the turn. At this point it's set to 0, since the incrementing instruction is at the top of the mainline flow. The next instruction, GOSUB 2000, will call up the subroutine that generates the computer's hidden number.

The logic of this routine is rather rote, but in essence the outcome will be three different integers stored in *Table A*. Using the system's random-number function the integers are fetched one at a time, and successive duplicates are refused, the computer insisting that another be called. Multiplication by ten in the RND expression will shift the offered number one position to the left. The result at that point will be an integer followed by five decimal digits. Construction of the expression with the INT (for *int*eger) function will truncate the decimal portion also, leaving only a value between zero and nine, inclusive.

Back to the mainline. In rapid succession, starting at line 140, there are three subroutine jumps to control sequential access to the major modules of the program. The balance of the mainline, lines 170 through 280, test for the end-of-game condition, permit optional replay, or terminate with the closing quip "SO LONG THEN."

The subroutine starting at line 3000 is wholly concerned with entry of a guess. It is here that the operator is prompted by the word *GUESS*, and with a wee bit of intimidation by showing the turn number with each prompt. The conditional in line 3020, read as *if equal to or less than 999,* will go around the "too big" message. A nested subroutine jump is then effected to line 3300 to parse the player's entry (as shown in Fig.A-2).

Upon return from the parsing subroutine the flow continues with the duplicate-digits test sequence that was described earlier. If all is copacetic the return instruction in line 3090 will link back to the

mainline. Notice that there is no escape from the player-input module until a guess is input that is satisfactory to the logic of the program.

The business of getting the clues is next, and it's all done by the set of tasks beginning at line 4000. This module is actually coded as a miniprogram itself, complete with a short mainline of its own to manage access to the three types of test necessary. Return to the game's mainline is from statement 4030.

Counter M1 is set to zero at line 4400 just before entering a FOR-NEXT loop that is allowed to execute a total of three times. Loop control uses the letter I, and the value there is also used as a subscript for comparing the digits in the two tables, *A* and *B*, in a parallel manner. The expression at line 4430 will increment the value of *M1* only in the event the table pairs being tested by line 4420 are equal. Notice also that later, upon return to the game's mainline, if M1 contains a three the game will end because the player's guess matches the computer's number.

The *astute* counting sequence is a finite loop, always repeating three full times, and a fallthrough occurs to the *abstract* tester which begins at line 4450. Another counter, M2, is set to zero, and three more tests are made through the use of a FOR-NEXT loop. Symbol I is again used for both loop control and as a table-accessing subscript. The sequence-of-three statement beginning at line 4470 compares each digit of *Table A* with each in *Table B*. *Any match* encountered here will preclude access to statement 4480, which counts mismatches (abstracts). The return instruction at line 4500 exits back to the minimainline for an immediate jump to the last set of tests.

The *askew* clue test routine is purely a linear sequence. After the M3 counter is initialized by statement 4600, the series of expressions that follows cross-compares the digits in *Table A* with those of *B*. For every instance of a *true* condition within this sequence a jump to the M3 incrementer at line 4680 will occur. After bumping down through all of these tests, with or without incrementing M3, the return at line 4670 will link back to line 4030. That statement also contains a return; so an exit is immediate, signifying that all tests are finished and it's time to return to the program's mainline.

And there—an immediate jump to line 5000 occurs (from statement number 160). This routing is a bit circuitous but with supposedly good intentions. Philosophically we suffer an inhibition about crossing major module block boundaries (statements in the *thousands* series). Sure *GOTO branches* could have provided direct

links, but the modular architecture would have been softened considerably.

The clue printing process begins at line 5000, and this is the last of the major tasks accessed by the mainline. A universal counter, symbol *M*, is used for controlling a repetitive print subroutine which is really a FOR-NEXT loop beginning at line 5500. The loop is allowed to execute *M* times, so the individual clue counters (*M1, M2,* and *M3)* are moved to M before the printer is called each time. A universal message field (M$) is also used by the print task, and it is initialized with the respective constant, ASTUTE ABSTRACT, OR ASKEW, just before it is invoked each time. After the last jump to the printing sequence a blank line is output by statement 5060, and a relink to the program mainline occurs from line 5070.

From that point on, the program either loops back through the mainline to reexecute the whole business or the game is over, depending on the outcome of the tests in lines 170 and 175.

THE PROGRAM

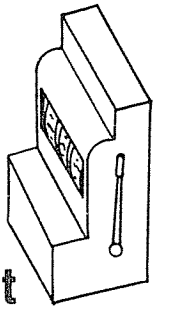
```
 10 REM  "ABSTRACT"
 20 REM
 30 GOSUB 9000
 40 DIM A(3), B(3)
 50 PRINT "YOU NEED INSTRUCTIONS (Y OR N)";
 60 INPUT Q$
 70 IF Q$ = "N" THEN 90
 80 GOSUB 1000
 90 PRINT
100 LET F$ = "I FORGOT TO TELL YOU THAT"
110 LET G = 0
120 GOSUB 2000
129 REM  "MAINLINE"
130 LET G = G+1
140 GOSUB 3000
150 GOSUB 4000
160 GOSUB 5000
170 IF M1 = 3 THEN 190
175 IF G < 20 THEN 130
180 PRINT "IT WAS:"A(1);A(2);A(3)
185 GOTO 240
190 PRINT "YOU GOT IT - ";
200 IF G < 13 THEN 230
210 PRINT "FINALLY!!!"
220 GOTO 240
```

```
 230 PRINT "IN ONLY" G "GUESSES!"
 240 PRINT "ANOTHER ROUND (Y OR N)";
 250 INPUT @$
 260 IF @$ = "Y" THEN 90
 270 PRINT "SO LONG, THEN."
 280 END
 999 REM  "INSTRUCTIONS"
1000 PRINT "I AM HOLDING A 3-DIGIT"
1010 PRINT "NUMBER...CAN YOU GUESS"
1020 PRINT "WHAT IT IS?"
1030 PRINT
1040 PRINT "WHEN YOU GUESS I WILL TELL YOU:"
1050 PRINT "   ASKEW = A DIGIT IS IN"
1060 PRINT "     THE WRONG PLACE"
1070 PRINT "   ASTUTE = A DIGIT IS IN"
1080 PRINT "     THE RIGHT PLACE"
1090 PRINT "   ABSTRACT = A DIGIT IS"
1100 PRINT "     INCORRECT."
1110 RETURN
1999 REM  "COMPUTER'S NUMBER"
2000 FOR I = 1 TO 3
2010 LET A(I) = INT(10*RND(1))
2020 IF I = 2 THEN 2070
2030 IF I = 3 THEN 2060
2035 IF A(3) = A(1) THEN 2010
2040 NEXT I
2050 RETURN
2060 IF A(3) = A(2) THEN 2010
2065 IF A(3) = A(1) THEN 2010
2070 IF A(2) = A(1) THEN 2010
2080 GOTO 2040
2999 REM  "PLAYER INPUT"
3000 PRINT "GUESS #"G"  ";
3010 INPUT @
3020 IF @ =< 999 THEN 3050
3030 PRINT "TOO BIG - CAN BE ONLY 3 DIGITS"
3040 GOTO 3000
3050 GOSUB 3300
3060 IF B(1) = B(2) THEN 3100
3070 IF B(2) = B(3) THEN 3100
3080 IF B(3) = B(1) THEN 3100
3090 RETURN
3100 PRINT F$
```

```
3110 LET F$ = "PAY ATTENTION"
3120 PRINT "MY NUMBER HAS NO 2"
3130 PRINT "DIGITS THE SAME."
3140 GOTO 3000
3299 REM  "PARSE PLAYER NUMBER"
3300 LET A1 = Q
3310 LET B(1) = INT(A1/100)
3320 LET B(2) = INT(A1/10)-B(1)*10
3330 LET B(3) = A1-B(1)*100-B(2)*10
3340 RETURN
3999 REM  "GET CLUES"
4000 PRINT
4010 GOSUB 4400
4020 GOSUB 4600
4030 RETURN
4399 REM  "COUNT ASTUTE CLUES"
4400 LET M1 = 0
4410 FOR I = 1 TO 3
4420 IF A(I) <> B(I) THEN 4440
4430 LET M1 = M1 + 1
4440 NEXT I
4449 REM  "COUNT ABSTRACT CLUES"
4450 LET M2 = 0
4460 FOR I = 1 TO 3
4470 IF A(1) = B(I) THEN 4490
4472 IF A(2) = B(I) THEN 4490
4474 IF A(3) = B(I) THEN 4490
4480 LET M2 = M2+1
4490 NEXT I
4500 RETURN
4599 REM  "COUNT ASKEW CLUES"
4600 LET M3 = 0
4610 IF A(1) = B(2) THEN GOSUB 4680
4620 IF A(1) = B(3) THEN GOSUB 4680
4630 IF A(2) = B(1) THEN GOSUB 4680
4640 IF A(2) = B(3) THEN GOSUB 4680
4650 IF A(3) = B(2) THEN GOSUB 4680
4660 IF A(3) = B(1) THEN GOSUB 4680
4670 RETURN
4680 LET M3 = M3+1
4690 RETURN
4999 REM  "PRINT CLUES"
```

```
5000 LET M = M1
5010 GOSUB 5100
5020 LET M = M2
5030 GOSUB 5200
5040 LET M = M3
5050 GOSUB 5300
5060 PRINT
5070 RETURN
5100 LET M$ = "ASTUTE"
5110 GOTO 5500
5200 LET M$ = "ABSTRACT"
5210 GOTO 5500
5300 LET M$ = "ASKEW"
5500 IF M = 0 THEN 5540
5510 FOR I = 1 TO M
5520 PRINT M$
5530 NEXT I
5540 RETURN
9000 REM  "RANDOM NUMBER ROUTINE"
9010 LET Z = RND(1)
9020 RETURN
```

B



Bandit

Maybe you have been to Freeport, Monte Carlo, or Las Vegas. If so you're undoubtedly familiar with slot machines. In any event we have here the poor programmer's facsimile of the one-armed bandits common to the gambling capitals of the world. Of course few microcomputers can cough up cold cash so a little pretending is helpful.

To partly compensate for your machine's inability to heap piles of winnings at your feet the game of *Bandit* does permit you to vary the amount of the bet with each figurative pull of the handle. And during the course of play an occasional whimsical message is printed, based on your winnings or lack thereof. Both are interesting features that the makers of real slot machines perhaps ought to consider adding.

In the absence of fancy graphics capability, instead of displaying pictures of grapes, lemons, and cherries, this program will print the names (GRAPES, LEMONS, or whatever). It does help, then, for the players of *Bandit* to at least be literate. The tidbits of humor that occasionally surface are more enjoyable also to the person able to read simple English.

This game is essentially mechanical. When the prompt appears (BET?), type a number to represent a dollar amount. The rules caution on the limit of $200 for a single bet. When you terminate the entry—by use of the RETURN, LINE FEED, or whatever your *fini*-key is called—the program will roll the three picture tumblers.

Your bet is lost if you don't get at least a pair of something. Any odd pair will pay you, though—twice whatever you bet. If all three

pictures come up alike your winnings for that roll are five times your bet—with the exception of BAR or JACKPOT. Triple bars are worth ten times the amount you bet, and triple jackpots will pay your bet twenty times over.

Opportunites to quit the game (or to elect to continue) are announced whenever your winnings or losses exceed $200 or whenever you are again even with the board. Notice also that it is not necessary to type an amount for each pull of the handle. The computer will remember the amount typed in at any point and will allow you to "let it ride" with each simulated pull of the handle, providing you terminate without typing anything at all. Points concerning the end-of-game boundaries and the automatically repeated betting are seemingly unrelated, yet they do have problematic implications.

Betting in hundred-dollar increments or similarly large amounts will indeed cause a short game. Conversely, nickel or dime bets may never exceed the $200 win or loss threshold; indeed, it may well take more than just all night for you to get back to zero—especially if you arbitrarily vary the bets.

Because of the way this all works it can be rather fun to force an end-of-game situation by blowing the $200 upper limit by intentionally losing more than $200 or, even more arduous sometimes, by trying to zero out.

DEFINING THE PROBLEM

It isn't at all difficult to envision the more or less simplistic sequences of conditional testing that can be devised to handle $200 entry and end-of-game business—nor even, for that matter, for the zero-balance aspect and the auto-repeat betting feature.

The real crux of this programming problem has to do with the printing of picture names from a list of ten or so, and in a seemingly circulating fashion, yet trapping at random points in the list. Obviously not only must the list be rolled three times, comparisons must be made to know when pairs or triplets occur; and in case of BAR and JACKPOT, at least, distinction is necessary because different win multipliers must be used.

DESIGN STRATEGY

Going for the simple direct solution could mean simply setting up a series of *picture-name* print statements and conditionally accessing them based on the use of a random integer. Of course the three random picks would have to be saved for the comparison tests. And every print line could be followed by a RETURN or by a GOTO that

branched to a common return statement so that the whole sequence could be treated as a subroutine—to satisfy the three-times usage requirement.

This could be made to work, but there is the monotony of coding ten print statements alternated with an exit of some type. And too, there's the unstated but certainly desirable need to print the three pictures on the same line, uniformly spaced apart. Tabbing could be used for that, but a looping technique would certainly be less tedious than using a long linear sequence for printing the pictures.

If the picture names were stored as elements in a table they could be fetched using the random number as a subscript. The table-accessing statement would move the constant to a general-purpose field for the print statement's benefit. These tasks and a tabbing mechanism could be included within a loop sequence. Even the three-times comparisons could be integral to the same loop. This scheme is definitely less tiresome to code than the previous idea.

The purist could introduce arguments at this point having to do with speed and storage tradeoffs, even including the technique finally adopted by *Bandit*. And our example would lose, at least by reason of a speed disadvantage. But this is a game program, and it is supposed to be a reasonable imitation of a slot machine. The method chosen for *Bandit* does use a loop, in fact two of them, one nested within the other. The inside loop manages the circulating list of ten picture names; but rather than a table, the list is a data-statement structure. A READ X$ is contained within the loop and a random value is set up to define the limit of FOR-NEXT iterations.

The outside loop provides the random number, saves that value, and does the printing of *X$*. The TAB variable is also generated by the outside loop arithmetically from the loop counter itself.

Successive data reading is a relatively slow technique, and there is a degree of randomness as to the time required since *X$* is located by use of a random number. The result is an impression of coasting, especially since the printing and reading tasks are interlaced within nested loops.

THE INTERNALS OF BANDIT

There is not a lot of modularity to this program, but what is there is shown by the template in Fig. B-1. As usual, the instructions to the player are self-contained in the subroutine that starts at line 1000. Statements 3000 through 3150 contain the instructions for rolling the tumblers as well as the picture names themselves.

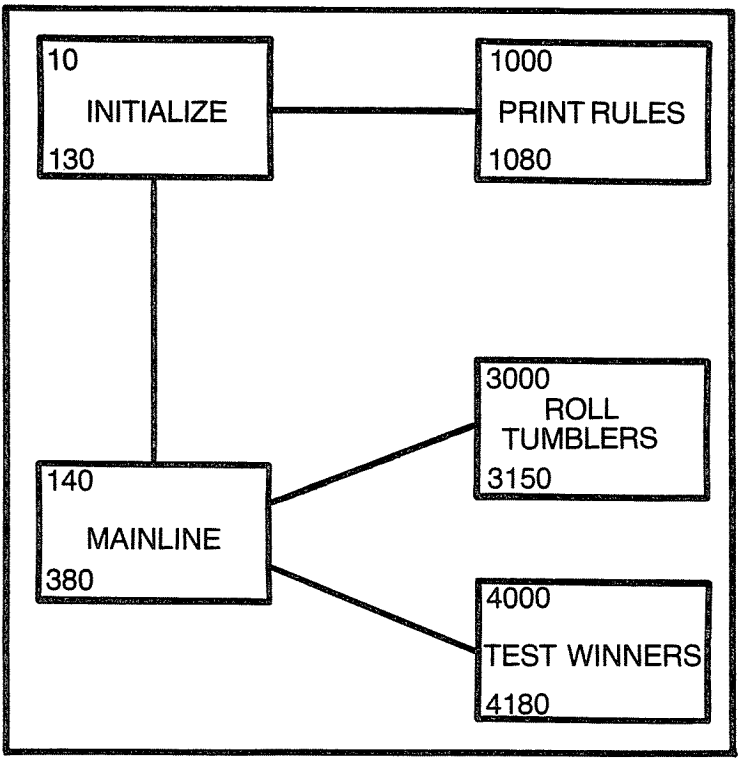


Fig. B-1. Program template for Bandit.

The other major task, computing the winnings based on pairs and triplets of pictures, is all done by the subroutine beginning with line 4000.

Symbol *R* is used extensively throughout the program and the mnemonic connotation of *R* is random. The DIM statement in line 30 sets aside a three-element table called *R* to temporarily store the value of each picture as it is rolled. The simple variable *R* is also a work field, used to store the random number that will trap the selected picture for each tumbler.

A lot of use is made of the number handed up by the random-number expression contained in statement 3060. The +1 tacked onto the end of the expression will cause a net value in the one to ten range. The *RND* function produces an integer between zero and nine, inclusive, but adding one gives a number from a series that starts with one.

Each picture is obtained by the FOR-J inside loop in lines 3080, 3090, and 3100. The READ X$ stops when *R* is reached, and the

RESTORE in line 3110 will reset the data pointer for use the next time.

The number in *R* at that point is also saved in *table R*, using the *I* loop counter as a subscript. This is accomplished by the instruction in line 3070, which you'll notice is outside of loop *J* but within loop *I*.

All of the comparison logic depends on the numbers contained in *table R*. If all three fields added together total three, they must each contain a one (JACKPOT), since zero can't occur. Similarly, if the sum is thirty (BAR), each roll was a ten, which is the largest number possible. Thus it is learned by statements 4010 or 4020 whether the bet should be multiplied by either ten or twenty and an appropriate branch is taken.

If the down-total of the table is neither three nor thirty the loop beginning at line 4030 is entered. Here it is learned whether trips of anything else exists. If so, the loop falls through to multiply the bet by five in statement 4060.

During the loop, though, any mismatch will cause a branch to the series of tests beginning in line 4120. This sequence will detect a pair, and any found will branch to 4170 to double the bet. Otherwise, the player's bet is lost by converting *Q* to a negative amount by subtracting the bet from zero.

Upon return to the mainline whatever is in *Q* is added to *T*, which contains the running total as a positive or a negative value, depending on whether the player is winning or losing. Immediately after updating the player's balance in line 260 a zero balance is tested for. If the balance is zero an option to stop is announced; otherwise, the player's account status is output at either line 340 or 370. Notice the use of the ABS (absolute) function in lines 340 and 350 to enable use of the balance without regard to the use of any arithmetic sign. And if 200 is exceeded the *stop* option is again allowed.

There are many opportunities to experiment with custom changes to this program. The whole business of multiplying the bets is subject to such modifications. Choices here are simply the author's, but they are related to the auto-repeat feature. There is a tendency by the player to let the bet stand, so the frequency of the messages output is somewhat controlled by either a series of wins or losses.

Since Q is itself factored (or made negative), and in our model will remain unaltered by a null input, either the bets are compounded or, more often, the operator attempts to bet a negative which causes an immediate message that they must enter an amount to play. And even though fractions of dollars are permitted by use of

decimal entries, a bet amounting to less than a dollar accuses the player of being a cheapskate.

All of these tests and messages are done in the mainline programming sequence, which can be totally rewritten if desired without affecting the logic of the other modules. It's also worth mentioning that none of the modules is dependent upon any housekeeping elsewhere.

The technique used for variable tabbing in statement 3120 might also be noted, since this trick is often useful for a variety of programming tasks. By arbitrarily multiplying the loop counter (an integer) by ten, then immediately subtracting ten, the values of zero, ten, twenty, etc. can be derived. This gives uniform spacing with the distance dictated by the multiplier.

THE PROGRAM

```
 10 REM  "BANDIT"
 20 REM
 30 GOSUB 9000
 35 DIM R(3)
 40 PRINT "WANT A DESCRIPTION (Y OR N)";
 50 INPUT Q$
 60 IF Q$ = "N" THEN 120
 70 IF Q$ = "Y" THEN 110
 80 PRINT "PLEASE ANSWER: Y FOR YES"
 90 PRINT "  OR N FOR NO..."
100 GOTO 40
110 GOSUB 1000
120 LET T = 0
130 PRINT
140 PRINT "BET";
150 INPUT Q
160 IF Q > 0 THEN 190
170 PRINT "YOU HAVE TO BET TO PLAY!"
180 GOTO 140
190 IF Q < 201 THEN 220
200 PRINT "BE REASONABLE NOW!"
210 GOTO 130
220 IF Q > .99 THEN 240
230 PRINT "CHEAPSKATE"
240 GOSUB 3000
250 GOSUB 4000
260 LET T = T + Q
```

```
270 IF T < 0 THEN 340
280 IF T > 0 THEN 370
290 PRINT "READY TO QUIT (Y OR N)";
300 INPUT Q$
310 IF Q$ = "N" THEN 130
320 PRINT "*** SO LONG, CHICKEN ***"
330 END
340 PRINT "YOU OWE ME $"ABS(T)
350 IF ABS(T) < 200 THEN 130
360 GOTO 290
370 PRINT "YOU'VE WON:  $"T
380 GOTO 350
1000 PRINT "THIS GAME SIMULATES A"
1010 PRINT "  ONE-ARMED BANDIT (THAT'S"
1020 PRINT "  A SLOT MACHINE, YOU KNOW)."
1030 PRINT "BET ON EACH PULL - UP TO $200"
1040 PRINT "3 JACKPOTS PAYS 20 X THE BET"
1050 PRINT "3 BARS PAYS 10 X THE BET"
1060 PRINT "3 OF ANYTHING ELSE PAYS 5 TIMES"
1070 PRINT "2 OF THE OTHERS PAYS DOUBLE"
1080 RETURN
3000 REM  "FETCH THE FRUITS"
3010 DATA JACKPOT, BELL, PLUM
3020 DATA ORANGE, LEMON, GRAPE
3030 DATA CHERRY, APPLE, MELON
3040 DATA BAR
3050 FOR I = 1 TO 3
3060 LET R = INT(10*RND(1))+1
3070 LET R(I) = R
3080 FOR J = 1 TO R
3090 READ X$
3100 NEXT J
3110 RESTORE
3120 PRINT TAB(I*10-10) X$;
3130 NEXT I
3140 PRINT
3150 RETURN
4000 REM  "FIGURE WIN AMOUNT"
4010 IF R(1)+R(2)+R(3) = 3 THEN 4100
4020 IF R(1)+R(2)+R(3) = 30 THEN 4080
4030 FOR I = 1 TO 2
4040 IF R(3) <> R(I) THEN 4120
```

```
4050 NEXT I
4060 LET Q = 5*Q
4070 RETURN
4080 LET Q = 10*Q
4090 RETURN
4100 LET Q = 20*Q
4110 RETURN
4120 IF R(3) = R(1) THEN 4170
4130 IF R(3) = R(2) THEN 4170
4140 IF R(1) = R(2) THEN 4170
4150 LET Q = 0-Q
4160 RETURN
4170 LET Q = Q*2
4180 RETURN
9000 REM  "RANDOM NUMBER ROUTINE"
9010 LET Z = RND(1)
9020 RETURN
```

C

Imagine—you just left the computer room heading for the canteen to have a cold soft drink. Enroute another programmer offers to flip a coin with you to determine who will buy. And as you round the corner in the hallway two people from another department offer to join you in this age-old form of office gambling.

Rather than risk being seen by a boss while gambling with a group in the corridors, suggest instead that a return to the computer room is in order. There you can introduce your friends to the game of *Cokes*, the computerized version of odd man out.

As provided here, this game permits from one to four persons to play. The computer picks a number between 1 and 999, and the players take turns trying to *guess what the hidden number is not.* Because there are so many numbers in the range from 1 to 999, even with four players the game could last all afternoon in this form. So there is another feature of the game that has been added to quicken the pace.

With each guess by a player the range permitted for the next guess is continuously reduced. If a player's guess is too high, the value of the guess is adopted by the computer to be the new top limit. In the same manner, when a guess is too low, the number just entered becomes the low-end limit for the next turn. Obviously, as the range narrows, the risk of bumping into the correct number increases rapidly.

For those players who are timid of heart but short on abstract reasoning capacity, you can discount the logic for an always-one-

less-than strategy. They're just as likely to encounter the hidden number that way as any other. In fact, you can hasten their play by using a strategy that roughly equates to a binary search algorithm; enter guesses that are halfway between the changing limits. You're certainly no more apt to accidentally pick the losing number this way since the odds are the same for all numbers regardless of the guessing scheme.

HOW THE PROGRAM PLAYS

Notwithstanding our enthusiasm for modular programming, as shown by a glance at the listing, this program is simple straight-line coding. Some programming problems just don't seem to deserve highfalutin engineering.

The program begins by asking how many players are to be entertained. The operator's response is stored at location P (for *players*); but before execution is allowed to continue, P is tested for a valid number in the one to four range. Naturally, less than one player is impossible, and more than four around a single console can cause too much commotion.

Another variable (T for turn) is preconditioned in statement 80, the only rule is printed by lines 85 and 86, and the game begins. The fatal number (to the loser) is picked by the random-number calling sequence of statements 100, 110, and 120.

The expression in line 100 will place a whole number in location X that is not larger than 998. Multiplication of the usual six decimal digits by 1000, and the *-1* appended to the end of the expression helps, of course. Zero and negatives are precluded by the test in statement 110. If the random number fetched into X by line 100 is zero or less the program forces a one into the variable at line 120.

The range limits are set up in *A* for the low and *Z* for the high. Lines 130 and 140 initialize the game with limits of 1 and 999, respectively. Thereafter, *A* and *Z* are updated as the game progresses. The input prompt line is of the form:

PLAYER # *t* GUESS *aaa* – *zzz*

where *t* is the turn (meaning player number), *aaa* is the low limit, and *zzz* is the upper limit. The three print statements beginning with line 150 are responsible for this output.

As soon as a guess is accepted by the instruction in line 180, it is masked by the INT (integer) function in line 190 to insure that location N will only contain a whole number. The player attempting to enter a fraction will soon get discouraged of such tricks since the

program simply ignores any decimal value by way of this masking technique.

If the tests in lines 200 or 210 detect an out-of-bounds entry the message "TOO LOW" or TOO HIGH is printed from line 300 or 320, and the same player must try again. Notice that the branching sequence is such that the turn number has not been altered, so the prompt line will repeat the player number.

Once execution gets past the range checks line 220 tests to see whether the guess is a loser—meaning N and X are equal. If not, and if the guess missed on the low side, a branch is taken to line 230 to update the lower limit; another branch continues on to line 250, where the player number is updated for the next turn. The alternate path possible out of the test in line 230 is a fallthrough, implicitly meaning the guess was on the high side. Location Z is immediately changed to whatever the guess was, and a direct access to the turn number updating occurs.

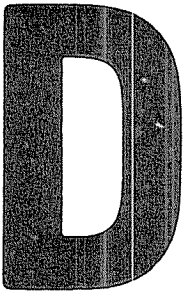
The turn number is incremented each time statement 250 is hit, but line 260 checks whether the value of P (number of players) is exceeded. If so line 280 is called into play to reset T back to 1. In either event the end of a complete round has occurred, and a branch from line 270 recycles the process, reentering at line 150.

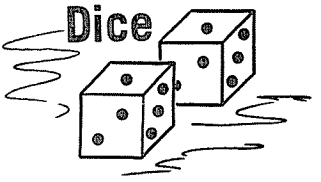
The whole game is just a loop in essence, and when finally a match on the hidden number is detected by line 240 an exit from the loop goes to line 340. Lines 340, 350, and 360 announce the loser, and a short looping sequence will print the numbers of the players that won.

THE PROGRAM

```
 10 REM  "COKES"
 20 REM
 40 PRINT "HOW MANY PLAYERS";
 50 INPUT P
 55 IF P<1 THEN 40
 60 IF P<5 THEN 80
 70 PRINT "TOO MANY PLAYERS"
 75 GOTO 40
 80 T = 1
 85 PRINT "GUESS A NUMBER WITHIN THE"
 86 PRINT "RANGE INDICATED"
 90 GOSUB 9000
100 LET X = INT(1000*RND(1))-1
110 IF X>1 THEN 130
120 LET X=1
```

```
130 LET A=0
140 LET Z=999
150 PRINT "PLAYER#";T;
160 PRINT " GUESS";A;
170 PRINT "-";Z;
180 INPUT N
190 LET N = INT(N)
200 IF N <= A THEN 300
210 IF N >= Z THEN 320
220 IF N = X THEN 340
230 IF N < X THEN 430
240 LET Z = N
250 LET T = T+1
260 IF T > P THEN 280
270 GOTO 150
280 LET T=1
290 GOTO 270
300 PRINT " TOO LOW "
310 GOTO 150
320 PRINT " TOO HIGH "
330 GOTO 150
340 PRINT "PLAYER#";T;
350 PRINT "BUYS THE COKES, GANG"
360 PRINT "FREE DRINKS FOR PLAYERS #"
370 LET I=1
380 IF I=T THEN 396
390 IF I>P THEN 410
395 PRINT I;
396 LET I = I+1
400 GOTO 380
410 PRINT
420 END
430 LET A = N
440 GOTO 250
9000 REM  "RANDOM NUMBER ROUTINE"
9010 LET Z = RND(0)
9020 RETURN
```





Here is a game that is especially suited to computers. In fact, although this game can be played with paper and pencil, much of the fun would soon dissipate because of the clerical tedium that is involved if it is played that way. Repetition and tedious arithmetic are the forte of modern *micros*, however, and it's easy to presume they are accurate and dispassionate as well.

The form of play is usual enough; two people taking turns rolling a pair of dice, and the score per roll is the sum of the numbers rolled. You are permitted multiple rolls per turn, and the score keeps mounting; but there is a catch. If a roll comes up as a match, exactly, with a previous roll in the same turn, you forfeit the turn and the score that was accumulating for that turn. Of course, you may relinquish your turn at any point voluntarily and receive credit for your score to that point.

There is another pressure-cooker element also. A game of *Dice* consists of but twenty turns each. The player with the highest score after twenty complete rounds is the winner.

PROBLEM DEFINITION

Up front, the housekeeping chores for a two player game are needed. Each will have separate scoring accumulators, and a mechanism is required to know whose turn it is. There is also the problem of limiting the play to twenty rounds, making sure that *both* players get their share of rolls.

The need for a temporary accumulator is apparent—needed to hold the score as it accumulates within a turn—but a single worker should suffice since it can be used for either player in turn. Maintenance of this field has to include (1) setting it to zero at the beginning of a player's turn, (2) adding to it the sum of each roll and, (3) clearing it if a match forfeit occurs (when a player voluntarily gives up his turn, the contents must be added to the player's running total).

Let's consider next the business of the dice. A pair of random numbers are needed, each limited to a range of one to six. If the two integers of a roll are placed immediately adjacent all of the possible combinations will be found in the series; 11, 12, 13...64, 65, 66. This suggests a possible scheme then. A table of at least sixty-six elements to hold any roll specifically; the value of the roll can be used to access the table.

Actually, sixty-six permutations with a pair of dice is not possible. No die has zeros. The sequence of 01 through 10 can't occur; either can 20, 30, 40, 50, or 60. Nevertheless, sixty-six is the largest table that would be required to hold any possible permutation.

To better define that part of our programming problem having to do with the dice consider the following. We must fetch a pair of random integers and add their sum to a temporary worker. Their combined value (side by side) may be used to access a table as column and roll subscripts into an array; or the pair can be used as a whole-number subscript to vector into a table. In either case if the table check reveals that a repeat has occurred a match must be flagged. If the table shows that this is a first-time instance for this pair of numbers, the table must be marked in some manner to benefit subsequent checks for this pair.

There is a housekeeping task associated with this or any other table scheme. All of the elements of the table must be initialized to a known state at the beginning of each round of play. Whether the cleanup is done immediately before a round commences or after a round is finished is not important—but a clean slate is needed somehow.

If we pause a moment now and mentally mull over the total problem as described above we can intuitively surmise that the bulk of the program will have to do with managing an individual round of play. The sum of the lines of programming needed for a two-player game otherwise is bound to be less than for the business of the dice. As it turns out, that is the case. At least it is in the model provided here.

THE ARCHITECTURE OF DICE

Our theme regarding modularity, as practiced elsewhere in this text, has not been abandoned in *Dice*, but it is less obvious. Part of the indistinction stems from the use of module boundaries denoted by line numbers in even hundreds (rather than thousands). This method was adopted to preclude a cursory assumption that the structure of this program is the same as those others. *It is different.*

Notice that the programming template in Fig. D-1 indicates that module boundaries are sometimes crossed directly. GOTO statements, for example, may be from a 700-series line number to an 800-series line number.

There is another interesting structure in this program. The bottom of the mainline is never crossed. A round of play is managed wholly by a FOR-NEXT loop. But contrary to most such loops, *the NEXT instruction is never sequentially passed*; meaning the TO limit is never reached. Two exits from the loop are possible, one in the event the player voluntarily gives up his turn, and the other case is on a forced exit because of a matched roll.

A GENERAL-PURPOSE TECHNIQUE

At a point in this program it is necessary to combine the two separate numbers of the dice into a whole number. The requirement is certainly not unique because it can occur in a variety of programming problems. The solution provided in this example isn't necessarily unique either, but it does work sufficiently well, and the amount of coding required is minimal.

Suppose we roll a 3 and a 6. From this we need 36. Multiply the first digit by 10, thus 30. Now, add the 6 to 30, and presto, 36.

Obviously the same algorithm applies with more than two digits. If three integers had to be combined the first would require multiplication by 100 and the second by 10; then the three values can be added together. Notice also this technique is actually opposite that described in Chapter A for parsing whole numbers into separate integers.

HOW DICE WORKS

We begin the study of the program from the top. In line 30, which is the first of any significance, is a *dimension* specification for a 66-element table called D (for dice). By now the reader can undoubtedly anticipate the use of this table, but we'll postpone describing the mechanics for accessing it until the appropriate point in the program.

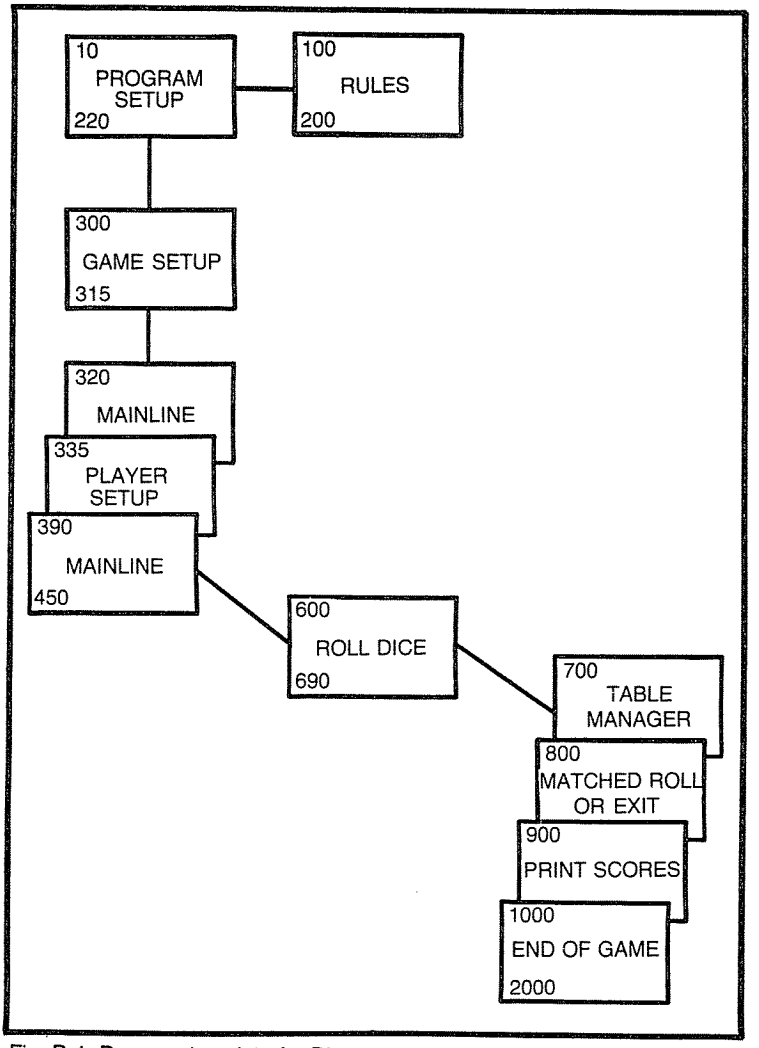


Fig. D-1. Program template for Dice.

Another table is indicated here, consisting of but two fields, and the table is symbolically called P (for players). The individual scores are kept in *table P* by use of a simple variable (also called P). A player number is maintained in P, either a one or a two, to be used as a subscript for addressing the scores in *table P*.

Perhaps you're curious as to the need for defining small tables by use of DIM statements. The model system on which these programs were built does provide for dynamic (implicit) dimensioning of tables of eleven elements or less (subscript range zero to ten).

As a general practice, though, you'll find that tables are always defined at the beginning of the programs in this book regardless of their size. This practice is valuable for documentation reasons.

A more important reason for always defining tables—even the short ones—has to do with debugging. Dynamic *re*definition of a table is not permitted by our model. Once a dimension statement has been encountered in a program you are not supposed to change the size of the table or array anywhere else within the same run.

Consider this. Suppose our problem logic requires a two-element table, just as in *Dice*. In the absence of a DIM statement a subscript of three (or four, five, six, . . . ten) will work mechanically even though it's contrary to our logical needs. If a mistake is made in a procedural area, creating a subscript variable larger than two, program execution will continue merrily on its way. On the other hand, a DIM D (2) will cause the system to flag any attempt to use a subscript larger than two. This can be a rather nice bit of assistance in the event a coding error stores values at table locations that were not supposed to exist by the program.

The operator is permitted the option of bypassing the description of the game. If the input to line 60 is Y lines 100 through 220 are skipped over. Either way statements 300 to 315 are gone through to set the scores to zero and to set a variable to one. The label M may mean monitor, master, mother, or whatever, but it's used as counter of the rounds of play—that's why it's started at one. When it goes past twenty the game is over.

Statement 325 checks whether the player number is two. Of course when the program first starts it should be zero, so P is made to be one in line 330. Later, P becomes two by adding one for the benefit of player Z's turn. Another pass through the same logic will add another one, making three, but this sequence will reset the player number to one. Thus, the player number (P) can never be anything other than one or two.

By the way, line 320 is the reentry point for successive rounds of play, and line 300 is the starting point for a new game. In line 320 the variable E is initialized for each round. This field is a program flag, E meaning *exit*. If we look momentarily at line 415 we see a conditional branch to statement 320 to start a new round of play in the event E is anything other than zero. The GOSUB 600 in line 410 breaks out of the mainline loop, and any task thereafter will upon a RETURN, allow the loop to continue if E is conditioned to zero; or a new round can be triggered by moving a value into E.

By similar logic, if M (the master counter) has reached twenty-one a new game is started by a conditional branch to line 220 from line 416.

Placement of these conditional exits is not because of sophistry. In a later routine it would be possible to branch directly to the beginning of another round or a new game. All other routines can only be gotten to by way of the GOSUB 600, however. If the integrity of the subroutine jumping is not maintained—if the RETURN sequencing is broken—the system's jump stack would eventually overflow. Imagine your chagrin if the program bombs out after several hours of play—for seemingly capricious reasons.

A round of play actually begins with the setting of the temporary score counter (S) to zero in line 335 and by clearing *table D* with the FOR-NEXT loop in lines 340, 350, and 360. Notice that only elements 11 through 66 are cleared—1 through 10 are never used. Neither are 20, 30, 40, etc. But why interrupt this compact loop for them? The input prompt at the start of a round reads:

PLAYER# p IS UP - ROUND# m

The rest of the mainline extends from line 390 through line 450. This is the FOR-NEXT loop mentioned earlier; notice that it is never exited by sequentially passing the NEXT statement. The TO limit is defined to be sixty-six, but there is no way it can be attained. A matching roll has to occur long before sixty-six is ever reached.

From within this loop the upcoming roll number is announced to the player, and a jump to line 600 goes to the dice roller. The usual RETURN from there will leave two numbers displayed, and the player will be asked "OK?". By preconditioning the response field (Q$) with an arbitrary Z a simple termination of the INPUT in line 430 will result in a not-equal-to-X condition at line 440, and another roll is executed.

If the player chickens out, though, and types an X before terminating the input statement a jump to line 820 will credit his account with whatever has accumulated in the temporary worker S. From that area round counter M is incremented every other turn (by lines 830 and 835) and the player number is bumped up by one in line 840. If the game isn't over, the scores are printed and the exit flag (E) is incremented to signal to the mainline that a new player is up.

The remaining routine, the dice roller itself, is fairly simple. Two sequences, one starting at line 610, the other at line 640, will set up N1 and N2, respectively. These are forced to contain integers in the one to six range.

If the RND statements offer any other number GOTO statements will repeatedly branch back, insisting on a valid number.

The first number rolled, N1, is multiplied by ten, summed with N2, and the result is placed in N. The dice are printed by line 680, and the table is checked by loading *table R* with whatever is at the location of *table N*.

If the location is blank (zero), the round number (I) is stored at that spot by statement 710, and another roll is permitted.

Whenever *table D* is checked and the spot that is accessed does not contain zero, regardless of whatever is there, the indication is that this same pair (the combination of both dice) has occurred before. The number that is actually stored in the table tells when it happened. Statement 800 tells the player that he has zapped, and R tells which roll was duplicated.

The game-ending option is usual enough. When the M counter hits twenty-one the game is over. The final scores are printed, and the high score is the winner. Another game can be started by answering with a Y when asked. A relink to the mainline is triggered from a Y test, and a branch is conditionally activated from line 416 to line 220 to begin again.

Perhaps a final suggestion is in order regarding this program. Debug it thoroughly. It's bound to get a lot of use because *it really is fun to play*.

THE PROGRAM

```
 10 REM  "DICE"
 20 REM
 30 DIM D(66), P(2)
 35 GOSUB 9000
 40 PRINT "DO YOU KNOW HOW TO PLAY DICE"
 50 PRINT "TO SKIP RULES, TYPE Y";
 60 INPUT Q$
 65 PRINT
 70 IF Q$ = "Y" GOTO 300
100 PRINT "THIS IS A 2-PLAYER GAME AND"
110 PRINT "YOU TAKE TURNS.  MULTIPLE "
120 PRINT "ROLLS PER TURN ARE PERMITTED."
130 PRINT "THE SCORE PER TURN IS THE SUM"
140 PRINT "OF ALL NUMBERS ROLLED."
150 PRINT "YOU MAY END YOUR TURN WITH"
160 PRINT "AN 'X' AND THE SCORE ADDS"
170 PRINT "TO YOUR TOTAL, BUT..."
180 PRINT "YOU LOSE THE SCORE IF YOU"
190 PRINT "MATCH A PREVIOUS ROLL."
200 PRINT "HIGH SCORE WINS AFTER 20 TURNS"
```

```
220 PRINT
300 LET P(1) = 0
310 LET P(2) = 0
315 LET M = 1
320 LET E = 0
325 IF P = 2 THEN 335
330 LET P = 1
335 LET S = 0
340 FOR I = 11 TO 66
350 LET D(I) = 0
360 NEXT I
370 PRINT "PLAYER#";P;
380 PRINT "IS UP - ROUND#";M
390 FOR I = 1 TO 66
400 PRINT "ROLL #";I;
410 GOSUB 600
415 IF E <> 0 THEN 320
416 IF M = 21 THEN 220
420 PRINT "OK";
425 LET Q$ = "Z"
430 INPUT Q$
440 IF Q$ <> "X" THEN 450
445 GOSUB 820
446 GOTO 415
450 NEXT I
600 REM  DICE ROLLER SUBROUTINE
610 LET N1 = INT(10*RND(1))
620 IF N1 < 1 THEN 610
630 IF N1 > 6 THEN 610
640 LET N2 = INT(10*RND(1))
650 IF N2 < 1 THEN 640
660 IF N2 > 6 THEN 640
670 LET N = N1*10 + N2
680 PRINT N1;N2
690 LET R = D(N)
700 IF R <> 0 THEN 800
710 LET D(N) = I
720 LET S = S+N1+N2
730 RETURN
800 PRINT "ZAP...MATCHED ROLL#";R
810 GOTO 830
820 LET P(P) = P(P)+S
830 IF P = 1 THEN 840
```

```
 835 LET M = M+1
 840 LET P = P+1
 850 IF M = 21 THEN 1000
 910 PRINT "SCORES: #1=";P(1);
 920 PRINT "  #2=";P(2)
 925 PRINT
 930 LET E = E+1
 940 RETURN
1000 IF P = 1 THEN 930
1005 PRINT
1010 PRINT "END OF GAME"
1020 PRINT "FINAL SCORES:"
1030 PRINT "PLAYER #1";P(1)
1040 PRINT "PLAYER #2";P(2)
1050 PRINT "PLAY AGAIN (Y OR N)";
1060 INPUT @$
1070 IF @$ = "Y" THEN 2000
1080 PRINT "GOODBYE"
1090 END
2000 RETURN
9000 REM  "RANDOM NUMBER ROUTINE"
9010 LET Z = RND(0)
9020 RETURN
```

E

Here is a game program that makes use of one of the transcendental functions. Transcendental, according to Webster, means "incapable of being the root of an algebraic equation with rational coefficients," or "being, involving, or representing a function," such as *sine*. This program does use the BASIC function SIN, so the math majors in our midst should enjoy it.

For those who have all but forgotten nearly all of this here's a brief review of sine.

Again, from Webster, sine is defined as "the trigonometric function that for an acute angle in a right triangle is the ratio of the side opposite the angle to the hyptenuse." No kidding.

If all of this puffy phraseology doesn't jog your memory don't despair. Just be aware that the distance a projectile will travel can be predicted if the upward angle at which it was shot is known. Here, of course, we do ignore a few things such as velocity, wind resistance, the weight of the round, etc.

To explain this game to the elementary-school set an analogy can be shown while watering the lawn. With the nozzle aimed ahead, nearly horizontal to the ground, the stream from the hose hits the ground a few feet in front of you. Horizontal is zero degrees of elevation. If you aim directly overhead that's 90 degrees of elevation of the nozzle. You will get wet. An elevation of greater than 90 degrees is beyond straight up, and you may well find yourself hosing down your living room through an open window.

Range, and how it is affected by elevation, can be demonstrated with the garden hose also. With the nozzle adjusted to produce a thin stream, begin again at the horizontal. Show that, as elevation increases, the distance from you that the water will carry also increases. At a point you will learn the greatest distance you can achieve without having to move. That is termed, by the guys with the big guns, as *maximum range.*

The game *Elevate* simulates the shooting of a cannon at a ship. You are told the maximum range of the cannon, and a fictitious artillery spotter gives you the approximate range to the ship. You are supposed to guess the elevation required to hit the ship and enter the degrees on the keyboard. When you terminate the entry the computer will figure the striking point of your shot and advise whether you were over or short on the range—and by how many yards. The game continues until you sink the ship. Actually, a direct hit isn't necessary, either. Any hit within 100 yards of the ship will damage it so severely that it can be assumed to sink.

As you fire away, correcting your elevation based on the spotter's correction data, your shots are counted. The object is to use as few rounds as necessary to score a hit. See if your friends can do better.

HOW THE PROGRAM WORKS

Because of the relative simplicity of the programming problem, the architecture of *Elevate* is just straight-line coding. In fact it would seem somewhat contrived to attempt modularity where it isn't warranted. A cursory scan of the program listing shows that it is basically all a mainline and that there are no major subroutines.

The program begins with an option regarding the printing of the game's description. Notice that if the description is printed by the statements extending from line 70 through line 185 the program will halt at the INPUT Q$ in line 190. There are two reasons for this halt.

The video screen of the model on which this program was built can only accommodate sixteen print lines per page. If the program wasn't halted where it is, the first six lines or so of the information would be lost due to automatic scrolling. The second reason for the stop has to do with *human factors* engineering. Stopping the program at this point will permit the player to read the descriptive information without being distracted by the message that follows—which is really part of the start of a round of play.

Line 200 is the start of a game of *Elevate*, and this is the point of entry if the game's description is bypassed. From here an an-

nouncement is made regarding the maximum range of the gun and the distance to the target.

The expression in line 220 will concoct the random range value in a worker called R. Perhaps you recall the expression "my dear aunt sally, "the initial letters of which also stand for multiply, divide, add, and subtract. Whatever value the system generates in response to the RND function it is first multiplied by 3000; *then the* result is subtracted from 43000, giving a random range result in R.

The expression in line 240 is the last of the game's initialization. When the program is first loaded *S*, the shot counter, is automatically preconditioned. The placement of this instruction is for the benefit of players who want to play again and again.

The abrupt change in the statement numbering scheme from line 250 to 2500 denotes the logical boundary between housekeeping for the game and the loop reentry point for corrected guesses. This is the point at which the player is asked for an elevation figure. Whatever the entry it's stored in location E, and the shot counter is immediately incremented upon termination of the keyboard entry. Two conditional branches in lines 2530 and 2540 determine all that happens after that.

A normal entry should be a positive value, considerably less than 90 degrees. If the guess is less than 88, the branch from line 2530 is taken to the program area starting in line 4000. That's the valid guess path, which we'll describe shortly. But first, the *errors*.

A guess in the 88 to 92 range would be nearly straight up. The message printed by lines 2550 and 2560 announces the idiocy of the entry and suggests that the gunner ought to make a hasty departure before the shell returns to the point from whence it was fired. The sequence next printed looks like this:

.

.

.

.

BOOM!

Concentric FOR-NEXT loops are used to print the periods that simulate the descent of the round from overhead. Loop I begins in line 2570 and extends through 2610. The inside loop, J, is really a

do-nothing loop that executes 100 times between periods. This is just a timer to slow down the printing. This routine was coded for a TV screen, and without the J loop, line scrolling on the screen is too fast. On a serial printer output device statements 2580 and 2590 should probably be omitted.

Following the trail of ten dots the explosion (BOOM!) occurs, and a branch from line 2640 returns the program to line 2500 for another guess.

The other type of mistake is an elevation entry of more than 92 degrees—which is past straight up, or in effect, behind you. The THEN 2650 in line 2540 is executed for any entry greater than 92. When it happens, this message is printed:

THAT SHOULD MAKE YOU A HERO—

THAT ROUND MAY HIT CAMBRIDGE!

A conditional (C) is set to one by statement 2670, and the GOTO in 2680 returns the program for another elevation guess. At whatever point, later, that a normal guess permits the program to proceed to the area on line 4000, the subroutine at lines 3000 through 3050 is called upon if C is found to contain a one. The delayed message is :

NEWS FLASH!

CAMBRIDGE COW HIT

BY MYSTERY MISSILE

The regular program path is from the keyboard entry point to line 4000. Whether or not "news flash" is printed, the current entry is qualified one step further. If the guess is less than one a simple "ILLEGAL" is printed and a return branch to line 2500 is invoked. Otherwise, the shot is somewhere near reasonable, supposedly.

The E value is multiplied by two, divided by 57.2958 in statement 4030, and the result is placed in E2. Worker J is set up in line 4040 by multiplying the SIN function of E2, by the maximum range of 46500. The result to this point, in J, is subtracted from the randomly chosen range (R) by statement 4050, and the difference is placed in N.

The temporary number N is an actual distance, so D is set up by line 4060 to contain this result without the decimal portion by use of the integer function. Since D can be either positive or negative, the conditional test in line 4070 included the absolute function to determine whether or not the distance differential is less than 100 regard-

less of the algebraic sign. If so a hit on the ship is conceded, and a branch to the game-ending area is in order.

If the ranging error is greater than 100 yards the sign is allowed to influence whether the *over* or *short* messages are printed. The ABS form of expression is used to tell the player his distance error, since the sign isn't wanted as part of the printed output.

A bit of humor is attempted at the end if more than seven shots were used to score a hit. Either way, line 4130 outputs the *shot* counter and the play-again option is encountered in line 4170.

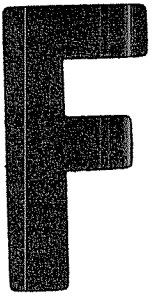
In any event this game is not very difficult to implement. It's fun for some players, and it offers lots of opportunities for experimental programming. So good luck and good shooting.

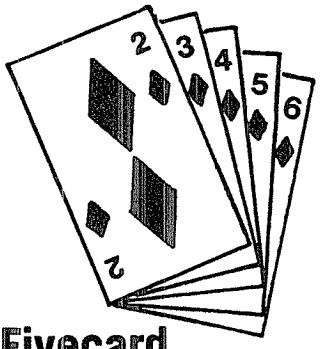
THE PROGRAM

```
 10 REM  "ELEVATE"
 20 REM
 30 GOSUB 9000
 40 PRINT "WANT A DESCRIPTION (Y OR N)";
 50 INPUT Q$
 60 IF Q$ = "N" THEN 200
 70 PRINT "THIS GAME SIMULATES THE EFFECT"
 80 PRINT " OF CHANGING THE ELEVATION OF"
 90 PRINT " A CANNON."
 95 PRINT "YOU ARE ON DUTY WITH A BOSTON"
100 PRINT " COASTAL DEFENSE BATTERY."
105 PRINT "YOU'RE TRYING TO SINK AN"
110 PRINT " APPROACHING WARSHIP."
115 PRINT "A HIT WITHIN 100 YARDS WILL"
120 PRINT " DO IT."
125 PRINT "FOR MORE RANGE:  ELEVATE."
130 PRINT "IF YOU OVERSHOOT:  LOWER THE"
140 PRINT " ELEVATION."
150 PRINT "ENTRIES ARE IN DEGREES "
180 PRINT ".....EXAMPLE:    45.3"
185 PRINT "OK"
190 INPUT Q$
195 PRINT
200 PRINT "MAXIMUM RANGE IS 46500 YARDS"
210 PRINT " DISTANCE TO THE SHIP IS"
220 LET R = 43000-3000*RND(1)
230 PRINT "      "R"YARDS"
240 LET S = 0
250 PRINT
```

```
2500 PRINT "ELEVATION";
2510 INPUT E
2520 LET S = S+1
2530 IF E < 88 THEN 4000
2540 IF E > 92 THEN 2650
2550 PRINT " YOU SHOT NEARLY STRAIGHT UP..."
2560 PRINT "...LET'S GET OUT OF HERE!"
2570 FOR I = 1 TO 10
2580 FOR J = 1 TO 100
2590 NEXT J
2600 PRINT TAB(15) "."
2610 NEXT I
2620 PRINT TAB(13) "BOOM!"
2630 PRINT
2640 GOTO 2500
2650 PRINT "THAT SHOULD MAKE YOU A HERO --"
2660 PRINT " THAT ROUND MAY HIT CAMBRIDGE!"
2670 LET C = 1
2680 GOTO 2630
3000 PRINT TAB(3) "NEWS FLASH!"
3010 PRINT "CAMBRIDGE COW HIT"
3020 PRINT "BY MYSTERY MISSILE"
3030 PRINT
3040 LET C = 0
3050 RETURN
3200 PRINT "ILLEGAL"
3210 GOTO 2630
4000 IF C <> 1 THEN 4020
4010 GOSUB 3000
4020 IF E < 1 THEN 3200
4030 LET E2 = 2*E/57.2958
4040 LET J = 46500*SIN(E2)
4050 LET N = R-J
4060 LET D = INT(N)
4070 IF ABS(D) < 100 THEN 4100
4080 IF R-J < 0 THEN 4200
4090 IF R-J > 0 THEN 4220
4100 PRINT "GOT 'EM";
4110 IF S < 8 THEN 4130
4120 PRINT " FINALLY!!"
4130 PRINT S"ROUNDS FIRED"
4140 PRINT
4150 PRINT "ANOTHER GAME (Y OR N)";
```

```
4160 INPUT Q$
4170 IF Q$ = "Y" THEN 195
4180 PRINT "END"
4190 END
4200 PRINT "SHORT BY" ABS(D)"YARDS"
4210 GOTO 2500
4220 PRINT "OVER BY" ABS(D)"YARDS"
4230 GOTO 2500
9000 REM  "RANDOM NUMBER ROUTINE"
9010 LET Z = RND(1)
9020 RETURN
```





Fivecard

The game program for this microcomputer version of poker pits man against machine in a variation of five-card stud. The rules are simple and the rounds are fast. The house and player start with an equal stake of $200. The ante for each hand is $5, and the betting limit per turn is $20.

Betting occurs only once in each hand, immediately after each player is dealt two cards. All cards are dealt faceup, and the player bets first. The computer-cum-dealer is expected to match whatever is bet. Then the remaining three cards apiece are dealt. Both hands are analyzed by the program according to traditional poker rules, the rankings are printed, and another round begins automatically.

The object of the game is for one player, man or machine, to win all of the other's money. The bet and the call are both subject to the $20 limit, but also to an account balance. When one or the other's stake falls below $20 the limit is reduced accordingly. The person can't bet more than he or she has left, nor bet more than the computer can cover if the machine's stake is nearly depleted.

As a game this program is entertaining, just as it is presented. It is also interesting as a program. We included it in this volume primarily because of the deck management scheme employed, and the poker-hand analyzer that it uses.

Programs that simulate card games must, in some way, hold a deck of cards in memory. In most cases it is up to the computer to do the shuffling and dealing as well. A number of techniques can be devised to do these things, but invariably a single concept prevails:

the deck is really a table or an array, usually containing numbers to denote both rank and suit.

At print time the codes designating the suits are converted to names, and the aces and the face cards are spelled out. The conversion is strictly for the benefit of the human eye; the internal numbering scheme facilitates using arithmetic for determing the relative ranking of hands.

The coding scheme used, the table structure itself, and the business of dealing and shuffling are all highly interrelated. There are two other card games in this book and they offer some contrasts to the methods used in this one. The following rationale describes why *Fivecard* is built the way it is, and why poker-like games insist that certain techniques be used.

FIVECARD DESIGN

Inveterate poker players have an intuitive sense of the order of probability in the drawing of hands. Any bias suspected of the machine's randomness will be quickly challenged. In fact, some will remain leery regardless of your contention that your program is honest. Let's see why.

One way to devise a card-dealing process is to generate a random number, force it into the one to fifty-two range, and reduce it to a rank and suit. To preclude an obvious *faux pas* it is also required that each succeeding card be compared with any previously dealt, ignoring any duplicates. There is an inherent risk to such a scheme. Some may contend that you are tampering with the natural odds.

The ultimate method is probably the one that accurately mimics a human dealer. Build a deck of fifty-two cards, thoroughly shuffle them, then deal one card at a time, alternating between the players and working strictly from the top of the stack, downward. In *Fivecard* we come close to such a scheme, save the part about dealing strictly from the top.

Instead of using random-number values to swap around the cards in the table (which is created as a sequential series), a pseudoshuffle is attempted by picking the cards from the table based on a random number.

Although this method may be vulnerable to an attack from the purist, it does seem to provide believable odds. The shortcut of not doing a preparatory shuffle is preferred in card games that require a new deck for each round of play. There are noticeable delays, as it is, and a *good* card-shuffling technique can require quite a few additional seconds.

There is another design factor imposed by poker and similar card games. Not only must the *rank* be of random origin but so must the suit. In fact, it's best to encode the deck so that the cards are each represented as complete values, with each having a serial value and a suit designation combined as one code. Some games pay no heed to color or pattern, so suits can be generated capriciously. The poker player is apt to holler if a *flush* crops up too often—you can bet on it.

HOW FIVECARD WORKS

The preceding discussion mentioned that the card deck is stored in a table. There are nine tables in *Fivecard,* but the deck is the largest one, and it has fifty-two numeric fields. The design of this whole program is based on the deck table, the other tables, and the use of subscripts to move values among them.

Before getting into those structures and their uses a couple of the design goals should be mentioned. Modularity is highly in vogue, so most of the major subroutines can stand alone functionally. This was done so that individual routines can be modified or rewritten completely without creating havoc elsewhere in the program. There is also the advantage that subroutines or groups of subroutines may be transported intact to some other program.

In essence the idea was to provide in *Fivecard* a collection of major task modules in a minilibrary of techniques useful to most card games. This is done through the use of two design methods. First, there is the game's character.

The identity of *Fivecard*, as seen by the player, is provided almost exclusively by the program's mainline. It is in the mainline that all of the dialog, operator input, betting, scorekeeping, and so on is done. Processes having to do with card-deck management, poker-hand analysis, and the printing of cards are all independent of the mainline, and most are independent of each other.

The subroutines, tables, and the mainline are kept independent of each other in two ways: (1) mainline procedures never access tables directly, and (2) for the most part one module is never allowed to tamper with another's table.

The tables, their individual structures, and their contents are described next. Since this whole program is based around these tables an understanding of the tables will make it easier to learn how the program works.

Table D

The card deck is stored in this table; its makeup is shown in Fig. F-1. As can be easily seen, there are fifty-two elements, and each

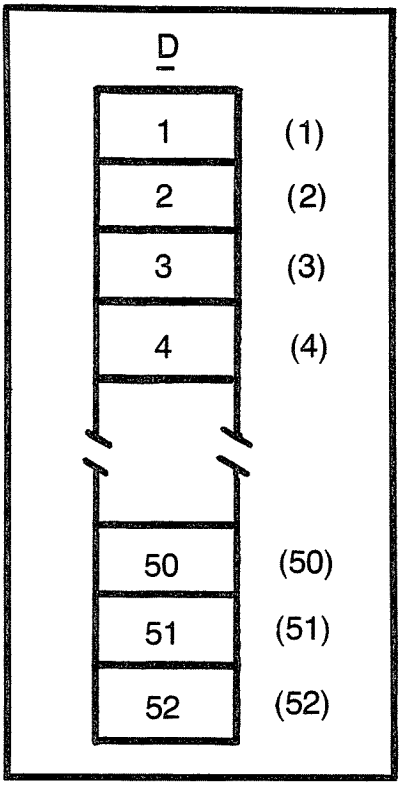


Fig. F-1. Table D, the deck.

contains a number from the set 1 through 52. The table is generated each time a new deck is needed. As cards are randomly picked during the deal, at the point the number is removed from the table, zeros are moved to that spot. It is this mechanism that precludes duplicate cards.

Tables A, A1, & B, B1

These four tables are each five elements deep, and they are used to hold the hands. *Tables A* and *A1* are for the human player; *B* and *B1* are for the house (the computer's hand). The structure of these four tables is shown in Fig. F-2. At the beginning of a round of play the hands are initialized to zero. As each card is dealt the card and suit codes are moved to the respective tables sequentially from the top.

Card Code Generation

A poker deck is supposed to have thirteen cards in each of four suits. The numerical deck in table D contains the numbers one through fifty-two. As a card is removed from the deck it is reduced to a number in the one to thirteen range and the suit is derived from

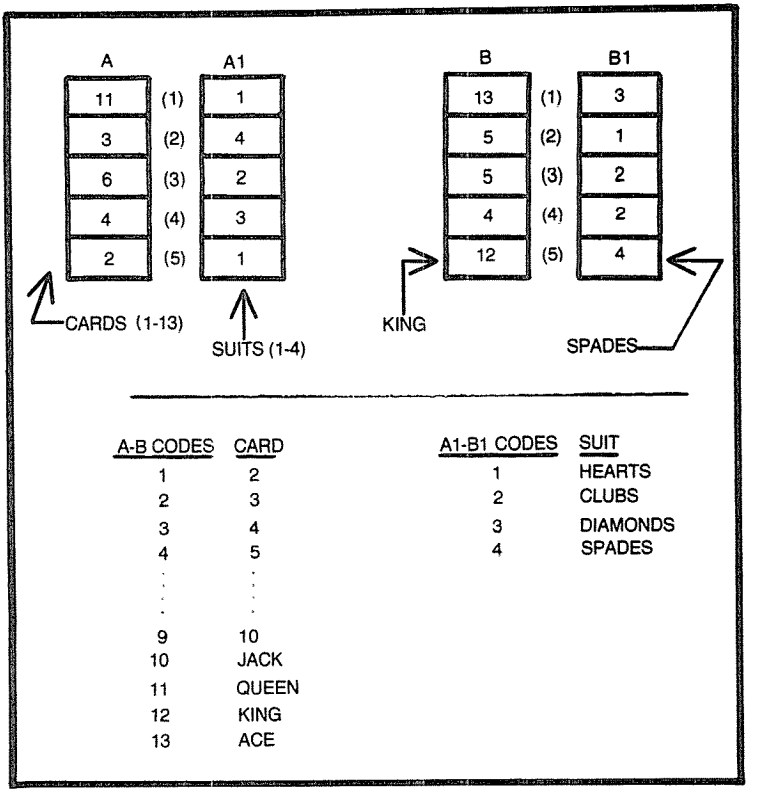


Fig. F-2. The players' hands.

the multiple factor. If the card as picked, is already in the one to thirteen category, the suit code is one. If the selected card number is larger than thirteen then thirteen is subtracted from it, and the result is checked for greater than thirteen. Each time the subtraction occurs a one is added to the suit number. At whatever point the one to thirteen test is satisfied the number in the suit counter becomes the code that is ultimately stored in either *table A1 or B1.*

Table H

The symbol H was chosen to mnemonically represent *hand.* This table is a general-purpose working storage area used by the *hand analyzer* logic. It's used for either player, depending on whose hand is being scrutinized. Since this same table is used for both players it is cleared to all zeros immediately prior to its use each time. Figure F-3 illustrates the contents of *table H* as it would appear during the analysis of the computer's hand, as it is shown in Fig. F-2 (*table B* and *B1*).

In poker one of the most significant aspects of a hand in determining its relative ranking is whether it contains pairs, triplets, or whatever. *Table H* is used to support an easy programming trick for counting duplicate cards in a player's hand. Each card number in a hand (*table A* or *table B*) is picked up, one at a time, and the number itself is used as the subscript for adding one to a counter in *table H*.

In the previous example (Fig. F-2, *table B*) the first card code is thirteen. A one is added to location 13 in *table H*, then. Since the table was cleared to zeros before this frequency-counting process began the one is added to zero the first time. As each card code is counted into the appropriate spot in the *hand* table several things can be determined about the player's hand by reviewing the contents of *table H*. Other things must be known ultimately, but here is what we can learn from the hand-counter table:

- The highest numbered location that isn't zero in *table H* tells the rank of the highest card in the player's hand. It's the location number that tells, not the code that's there.

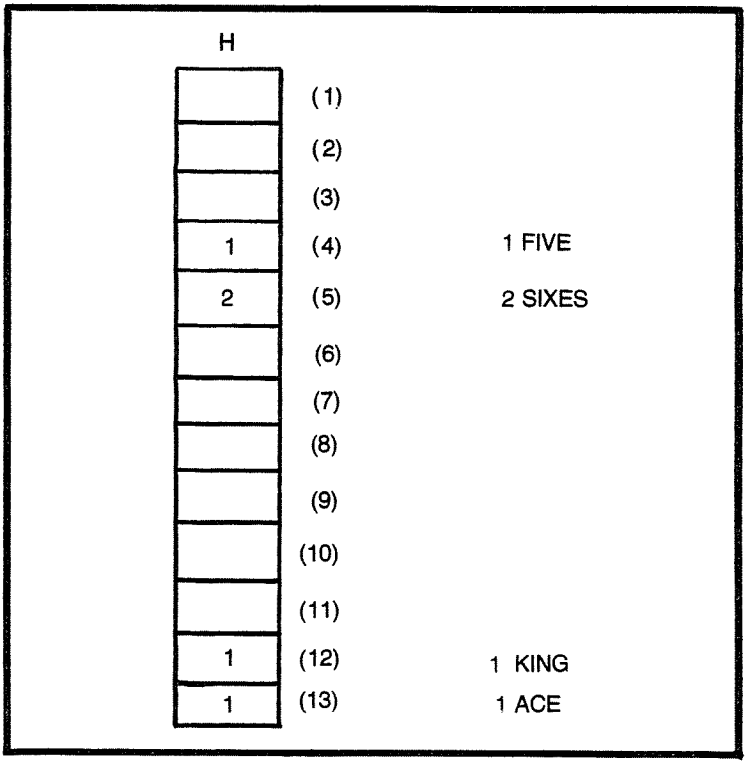


Fig. F-3. Table H, the hand count.

- Multiples can be detected by examining the numbers at any *nonzero* location. A two is a pair; three is triplets, etc.
- The card value of multiples is the same as the *table H* element number (plus one). In the example the count of two in location 5 will eventually decode to mean a pair of sixes.
- If none of the counters is larger than one the hand cannot be a winner on the basis of pairs, three of a kind, or anything of that type.
- Straights can also be deduced from this table. One way is to scan from the top, incrementing a counter for each one. Once counting begins the first non-one breaks the counting process. At that point if the count is five the hand holds a straight. Anything other than five contiguous 1s is zip.

It is incidental to *Fivecard*, but for the same reason *table H* knows the player's highest card, the lowest card is also known. As can be seen, a considerable amount of data can be derived from *table H*, yet the mechanics for doing the frequency counting is quite simple. Suits must be looked at as well in poker, so additional tables are used for the total analysis of the player's hands.

Table R

As can probably be guessed, R means rank. It too starts as all zeros, is built on the fly during the analysis, and is used temporarily for either hand. Figure F-4 has a picture of *table R*, and the numbers in the illustration reflect the same example (*table B*, Fig. F-2). Obviously, the depth of the table (ten elements) coincides with the ten possible poker rankings.

The buildup of *table R* begins by making several passes down through *table H*. In the first pass the hand table is searched for a straight. If this search reveals five ones with no intervening zeros a code of five is forced into *table R at location 6*. Otherwise , this spot will be left with nothing in it, from which we may infer that the hand does not contain a straight.

In another pass through the hand table multiples are looked for. During this scan any number larger than one indicates the player's hand does contain matching card values. The serial value of the matching cards corresponds to the *table H* subscript address at which the multiple's code is found. The card value is then moved into *table R*, according to the number of duplicates that there are for that value.

It is in this manner that *table R* is made to show quads, triplets and pairs. If a hand contains four sevens then seven is stored in *table R* at location 3. The third spot from the top is used because four of a

RANK	CODE	R	
ROYAL FLUSH	HIGH CARD		(1)
STRAIGHT FLUSH	HIGH CARD		(2)
4 OF A KIND	CARD VALUE		(3)
FULL HOUSE	TRIPS VALUE		(4)
FLUSH	6		(5)
STRAIGHT	5		(6)
TRIPS	CARD VALUE		(7)
2 PAIRS	HIGH PAIR VALUE		(8)
PAIR	CARD VALUE	5	(9)
HIGH CARD	CARD VALUE	13	(10)

Fig. F-4. Table R, the ranking of a hand.

kind in poker is the third highest possible hand. The card value (seven) is stored there so that if the opponent's hand also has four of a kind, the one with the higher number can be declared the winner.

This same convention is used for triplets and pairs. *Table R* locations 7, 8, and 9 are used, respectively, to denote three of a kind, two pairs, or a single pair. In the earlier example two fives can be seen in Fig. F-3; these will eventually be revealed to the players as a pair of sixes.

After the scanning of *table H* for multiples is completed if the ranking table reveals that two pairs are present, the higher-ranking pair is coded in *table R* in field *8*; the other pair is noted in field 9. Logic within the *multiple* scanner will cause the higher of two pairs to bubble up from field 9 to 8. If only a single pair is found it's noted in field 9 and 8 stays empty.

The rank table is itself examined several times to learn more about the player's hand. If a triple and a pair are noted the lucky player has a full house; the card code for three of a kind is placed in the fourth spot down in *table R*. Here again if both player's get a full house, the winning hand can be argued on the basis of which has the higher number in location 4 of *table R*.

The last spot in *table R* contains the numeric equivalent of the highest card in a player's hand that is not part of a matched set. If a

tie breaker is needed this is the card used. By the same token, if *table R* is scanned from the top down and found to be all empty location 10 will show us a high-card hand.

The values in *table R* at this point are used in conjunction with a suit scanning process to further decide whether more codes need to be developed within the table. Card suits are only significant when they are the same for all five cards in a player's hand. As far as the game's internal logic is concerned it doesn't matter which suit is held. It is only significant if they're all alike.

Table A1 or *B1* is examined to determine flush conditions. If a five-of-a-kind condition exists in either table, we have a flush. A code of (six) is generated to denote a flush, and it is stored in location 5 of *table R*. If there is a code at both locations then, subscripts five and six, a straight flush can be deduced. If so the high card of the straight (from field 6) is replicated in the second location down in *table R*. The same scheme works for that one-in-a million maxi: the royal flush. Needless to say the top spot in *table R* isn't encoded often.

Tables A2 & B2

These are the remaining tables, so this overview is nearly over. There isn't any illustration this time. *Tables A2* and *B2* are really duplicates of *table R* (Fig. F-4). Once the human's hand (player A) has been codified the values built up in *table R* are copied into *A2*. The same is true for the computer-cum-dealer (player B): *table R* is copied into *B2*. A parallel comparison process can be made after both hands are analyzed by cross-checking the corresponding locations of *A2* and *B2*.

Most of the time a winner is readily determined by scanning from the top of each of these tables. The first non-zero condition usually marks the winner—assuming the other table has a zero at the same spot. Ties have to be further argued. If one number is larger than the one at the same spot in the other table the smaller one is the loser. If the two numbers happen to match a tie condition must be broken if possible.

One device for breaking ties relies on the sum of the values in each of the tables (*A2* and *B2*). A comparison of their totals will break most ties; the exception is when their respective high cards are the same, in which case it's necessary to sum the original hand tables. This trick will account for any discrepancies in the value of the second highest card, etc.

The mechanics of comparing *table A2* with *B2*, and subsequent tests for breaking ties, are all separate from the hand-analyzer

routine. We will delay further discussions on this until you are well into the internal workings of the program.

As a brief review of the tables: *A, A1, B,* and *B1* are the player's hands as dealt. *Table H* is dimensioned at thirteen, each spot being used as a counter location for each of the thirteen card-code numbers possible. *Tables R, A2,* and *B2* are all alike—ten spots—for holding ranking codes.

NOTES ABOUT THE ARCHITECTURE

It has already been mentioned that *Fivecard* is modularized. The proponents of *structured programming* may wince at this claim, but perhaps they are not fluent in BASIC. To argue: there are discernible boundaries between functional modules, and most tasks are accomplished by stand-alone subroutines.

The programming template shown in Fig. F-5 illustrates how the tasks are broken out. Like most other programs in this book all

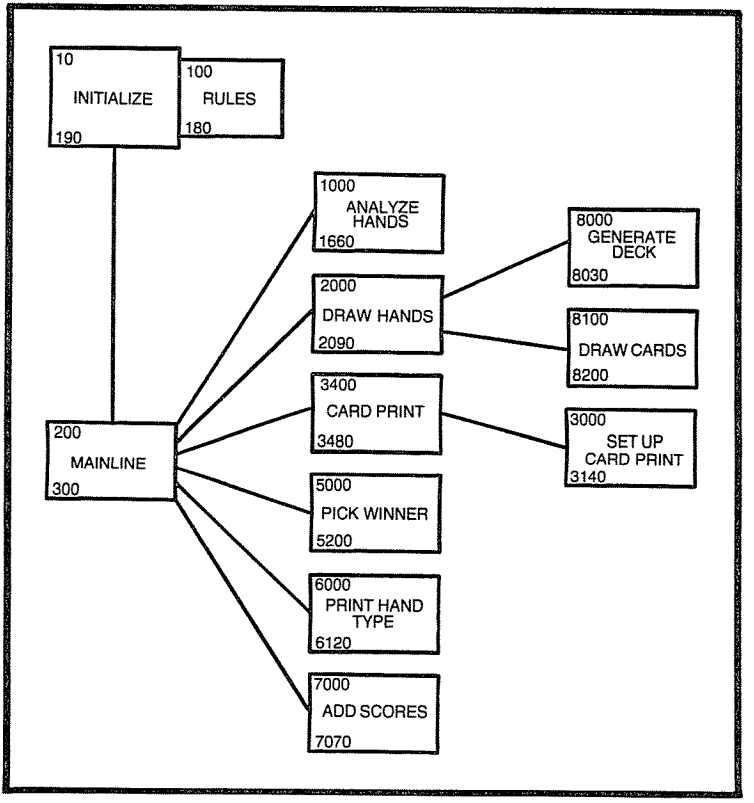


Fig. F-5. Program template for Fivecard.

of the global housekeeping, the rules for playing, and the startup are closely combined up front. Once the mainline structure is entered, GOSUB statements are used to call up the tasking subroutines in whatever order they are needed. And as usual capricious boundary violations are guarded against. In the mainline GOTO statements can only reference three-digit line numbers (which are exclusive to the mainline). Similarly, module-to-module linking is only permitted by exiting from one, back through the mainline, to get to another. That is, there are no direct branches between groups of statements denoted by whole thousands.

One more note is important: the size of this program may be a problem, depending on the vastness of your memory. Mine is a tight fit. In fact although the REM statements are few and far between they must be purged on our model, just to get the program loaded. There are other places where you may do some "shoehorning" if necessary. Some candidates are to cut out the rules—that's a lot of alpha code—and if that isn't enough, consider the lengthy descriptions in the card and hand-ranking printing tasks.

A primary philosophy of the architecture for *Fivecard* was to make it easy to leave out, add to, and overhaul subroutines, or modify the mainline, without wrecking the general fabric of the program otherwise.

HOW THE MAINLINE WORKS

The first thing worth noticing in the program listing is the table dimensioning statements in lines 30 through 60. The business of printing the rules and the option to do so are next, extending from line 70 through line 180. Which brings us quickly to the program's housekeeping area.

Initialization starts at line 200 by presetting two counters to $200. Counter Y is for you and counter M is for me, as considered from the machine's point of view. The symbol Q is unconditionally zeroed in this area also. Line 215 does this; it's necessary to get rid of any residue in this worker whenever a new game begins. The remaining replay item is to set the player counter (P) to one, as can be seen in line 230.

The restart point for each round of play is statement 240. From there, we have an immediate subroutine jump to the scorekeeping module (lines 7000 through 7070). It is within that subroutine that the stakes are maintained.

The scorekeeping process is relatively simple; at a glance it can be seen that the input worker is Q. The jump to the scorekeeping module happens at the beginning of each round of play, and that is

why Q had to be set to zero initially. At the start of a game a null update of the stakes occurs, but thereafter Q will contain the bet. Upon return to the mainline at statement 250, Q is again cleared in preparation for the next bet.

The conditional tests in lines 260 and 270 check whether either player is yet busted. If so a THEN expression exits to line 290 or 340 to close out the game. Otherwise, the game continues at line 360.

From statement 360 a chained jumping sequence occurs. The first GOSUB is to line 2000, the *draw* module. Another GOSUB is executed immediately from there to the deck-generating routine. Only four lines of coding are required there (lines 8000 through 8030) using a FOR-NEXT loop to create the number series one to fifty-two and to load them into *table D*. When the table is full, a RETURN gets the program back to the draw routine.

Uninformed players might be quite surprised if they learned what next happens internally. The loop sequence from line 2010 to 2090 will deal all five cards to each player (you and the machine). The actual draw is done by the task module that extends from line 8100 to line 8200. This subroutine is called alternatingly during the deal, setting up the cards in *tables A* and *A1* for the human, and in *tables B* and *B1* for the computer. Once both hidden hands are complete, a RETURN in line 2090 exits back to the mainline at line 370.

A two-times loop from line 370 to 390 includes a jump to line 3400 to effect the dealing of the cards visibly. Each iteration of the loop deals one card to each player. It's in this manner that two cards apiece are dealt preparatory to the betting dialog. The reason a FOR-NEXT loop is used, even though it's only done twice, is that the FOR-variable (I) is used as a subscript to fetch the card codes from *table A* and *table B*.

As revealed in the listing the sequence from line 3400 to 3480 is really a minimainline. A common pair of workers, C for card and S for suit, are loaded alternately from the hidden hands, and the dealing sequence is maintained in I. The values in C and S are carried to the print setup task (lines 3000 through 3140) to get the alphanumeric output, which is then displayed within the minimainline.

After two cards from each of the hands are revealed the mainline is continued from line 400. This simple prompt—BET?—is output, and execution halts for keyboard input. The sequence of code from line 420 to 500 edits whatever is typed into the numeric variable *Q*. The checking that is done is based on the assumption

that the operator is most likely to enter bets in terms of dollars—or at least as dollars and cents.

The routine betting limit of $20 is tested for in statement 450. The simple variable *L* is used in the limit tests also. The actual conditioning of the *L* variable is done by the scorekeeping routine, and that is why a GOSUB 7000 occurs from line 510 (and again at the beginning of a round of play). In the event one or the other player's balance has fallen below twenty, *L* is adjusted accordingly. The simple checking that is done does not preclude decimal entries, however, so those players that are not receptive to suggestive conditioning may enter decimal values, but at no special risk to the program's arithmetic integrity.

After the bet the balance of the hands are dealt by the sequence that extends from line 520 to 540. This one works the same as that for the first two cards, but this time the loop variable begins with three and runs through the fifth card. All that remains is to analyze the hands, compare them, and declare the winner. A pair of jumps (lines 550 and 560) analyzes the player's hand then prints the results; the computer's hand is done by the same modules, and that pair of jumps is in lines 590 and 600.

The GOSUB 5000 from line 620 takes off to the subroutine that does the actual comparisons to determine the winner of the hand of play. Upon return from there the end of the mainline is encountered in line 630 and a branch back to line 220 begins another round. Before the loop can be closed a lot goes on, though, and that is what is described next.

FIVECARD HAND ANALYZER

As the chairman of the Brotherhood of Butchers once said, ". . . and now folks, we come to the *meat of the matter.*" For us anyway, the meat of a poker-playing program is the business of figuring out what's in a hand. There are sixty-seven lines of code required to do it, extending all the way from lines 1000 to 1660; but it is not nearly as foreboding as a casual glance at the listing may imply.

Recalling the earlier description of the table structures, notice that the basic requirements are to do frequency counts of any matching sets of cards in a given hand, check for straights, check for flushes, and dump the tallies into either *table A2* or *B2*, depending on whose hand is being looked at. Not really so much after all as we shall soon see.

One reason this module appears to be lengthy is that it contains it's own housekeeping tasks, even including the table-to-table transfers. Another reason for its length is that there are several suppor-

tive tasks tacked on to the bottom of the module; it actually ends at line 1480 or 1520. The exit that is taken depends on whether *table A2* or *B2*, is finally set up. It does begin at the top and proceed downward, so that's how we will study it, but one task at a time.

Clear Table-H

A nice tight loop (lines 1010 through 1030) moves zeros into each of the thirteen elements of this table. Agreed, many are already blank. At most no more than five of these counter locations are ever used at one time. By arbitrarily zeroing them all none is missed, and it doesn't really matter which were used the last time through here.

Frequency Counts

Another loop, albeit not quite as tight, runs but five times to count the cards, including multiples. The loop itself is from line 1040 to line 1070, but it includes repeated jumps to the series of instructions from line 1620 to 1660. The supporting subtask controls the pickup of cards from either *table A2* or *B2*, depending on the P variable (one or two). With each iteration of the loop a one is added into *table H* at the appropriate spot, as determined by the code value of each card.

Clear Table R

Compactness, all the way. This is another three-line FOR-NEXT series. The result: zeros in all ten fields of the rank table. After execution of lines 1080 through 1100, ten times, a fallthrough is automatic.

Multiples & Straights

The whole of this sequence is from line 1110 to 1270. All but lines 1260 and 1270 are part of a FOR-NEXT loop that runs unconditionally for thirteen times. There are several internal conditional branches, however. Gradually being set up are *table R* fields, numbers 3, 6, 7, 8, and 9. In poker language these are: four of a kind, straight, triplets, two pairs, and a pair, respectively. It is worth noting that the loop's logic assumes that a straight exists until proved wrong.

During the loop all ones are counted into location 6, but when the loop ends at line 1260, if the count in *table R (6)* is not five, it's immediately zeroed. Either way, processing continues with line 1280, which is the start of another subtask.

High Card

Whatever the highest numbered card in a player's hand is it's forced into the tenth spot of *table R*. This is accomplished with the FOR-NEXT scheme, lines 1280 through 1310. The structure of this loop is interesting for two reasons: it counts backwards, and it never completes. The loop variable (I) is set to thirteen so that the scanning of *table H* can begin at the bottom. By incrementing with a *negative 1* each higher spot in the table is examined until something is found, ignoring all zeros.

The high card is actually a generated number. The I variable itself is moved into *table R(10)* with each cycle of the loop. When the conditional expression in line 1300 finds a not-zero condition, the THEN 1320 takes over, breaking the loop and abandoning whatever the count in *table R(10)* is at that point—which is, amazingly enough, the exact equivalent of the highest card in the player's hand.

Full House

This is the easiest task of all. Lines 1320 and 1330 look at *R(7)* and *R(8)*, and if they are both not zero, statement 1340 moves the value of the triple (from location 7) into *table R* at location 4. The residue below in *table R* can be left since later analysis works downward from the top of the rank table.

Flush

The first test is for a simple flush—meaning there is no consideration at this point regarding straights. A five-times loop begins at line 1350 and runs until line 1380, including a subroutine jump to the series at lines 1530 through 1610. Duplicate copies of the same logic occur there, one set each for fetching the suit codes from either *table A1* or *B1*—again based on whether P is one or two.

The all-are-alike qualification is done by comparing each card's suit code with the one at the bottom of the hand (fifth element). If any comparison fails the loop is broken with a THEN 1440, which completely bypasses the further testing for more perfect hands.

Straight Flush

The odds are getting tougher. In fact this program ran many hours before I was confident this test sequence worked. The test is simple, though. If a flush is present, as detected by line 1380 having resulted in a fallthrough, then the straight indicator, which is a code of five in *R(6)*, is tested for. If both conditions are true, meaning we have a straight and a flush, *R(2)* is loaded with the high-card value that was generated earlier in *table R(10)*.

One further test is required here. If that panacea of poker is present, meaning that not only do we have a straight flush but that *R(10)* contains a thirteen, a thirteen is slipped quietly into the first element of *table R*. This is supposed to be done by statement 1430, but I have yet to see it actually happen, at least in live play.

All that's left is to copy the contents of *table R* to either *table A2* or *B2*. The P value is checked in line 1440; if it is a two, the THEN 1490 takes over or a fallthrough takes place—and either way the hand analyzer is finished.

Perhaps you will now agree, although this module seems lengthy, it is actually quite simplistic in design. Notwithstanding the amount of time it takes to read about it and study it the onboard execution time is pretty reasonable. Even if we include the time to compare the hands it doesn't take long for either this narrative or the computer, as you will notice in the next and final section.

FIVECARD WINNER PICKER

Add up everything in the tables. That's the first step this module does, although for many hands the winner can be declared before these totals are ever needed. As can be seen in the listing lines 5000 through 5030 contain a simple nine-times loop to add downward through *tables A2* and *B2*, leaving the results in the high-card spot. This is done so that if the following tests find a tie situation higher up in the tables perhaps the sums of the tables will break that tie.

The next loop (lines 5040 through 5060) is broken whenever one or the other of the two tables has something other than zero in it. Scanning begins from the top, and the expression in line 5050 checks them in a parallel manner by adding together the corresponding fields in *tables A2* and *B2*. If other than zero comes up, one of them is not zero (or maybe both are not), so the program branches to line 5070 to learn more.

It is possible of course that both tables contain a code at the same distance down from the top. Line 5070 checks if they are the same values; if so a tie is in the making. Otherwise, the test in line 5080 determines which is larger. The winner and loser are announced accordingly. The results of the betting are displayed, and a return to the mainline occurs.

So goes *Fivecard*. The casual player may think this is simple game. Show them the listing. But it is fun, especially the program. The coding and loading did seem a bit of a chore, but getting it all to work proved very entertaining.

THE PROGRAM

```
 10 REM  "FIVECARD"
 20 REM
 30 DIM D(52)
 40 DIM A(5), A1(5), A2(10)
 50 DIM B(5), B1(5), B2(10)
 60 DIM H(13), R(10)
 70 PRINT "WANT THE RULES (Y OR N)";
 80 INPUT Q$
 90 IF Q$ = "N" THEN 190
100 PRINT "YOU AND I WILL PLAY 5-CARD STUD."
110 PRINT "THE ANTE IS $5.    BETTING LIMIT"
120 PRINT "IS $20.   I CALL ALL BETS AND WE"
130 PRINT "SPLIT ON TIE HANDS. WE HAVE $200"
140 PRINT "EACH.   GAME ENDS WHEN ONE OF US"
150 PRINT "GOES BROKE.   BETTING IS ALLOWED"
160 PRINT "ONLY ONCE - AFTER 2 CARDS APIECE"
170 PRINT "ARE DEALT.  THE DECK IS SHUFFLED"
180 PRINT "BEFORE EACH ROUND."
190 GOSUB 9000
195 PRINT
200 LET Y = 200
210 LET M = Y
215 LET Q = 0
220 PRINT
230 LET P = 1
240 GOSUB 7000
250 LET Q = 0
260 IF M = 0 THEN 290
270 IF Y = 0 THEN 340
280 GOTO 360
290 PRINT "I'M BUSTED"
300 PRINT "ANOTHER GAME (Y OR N)";
310 INPUT Q$
320 IF Q$ = "Y" THEN 200
330 END
340 PRINT "YOU'RE BROKE"
350 GOTO 300
360 GOSUB 2000
370 FOR I = 1 TO 2
380 GOSUB 3400
390 NEXT I
400 PRINT TAB(9) "BET";
```

```
 410 INPUT Q
 415 LET Q = ABS(Q)
 420 IF Q > 0 THEN 450
 430 PRINT "HUH?"
 440 GOTO 400
 450 IF Q < 21 THEN 480
 460 PRINT "HOUSE LIMIT IS $20."
 470 GOTO 400
 480 IF Q =< L THEN 510
 490 PRINT "THE LIMIT IS NOW ONLY $"L
 500 GOTO 400
 510 GOSUB 7000
 520 FOR I = 3 TO 5
 530 GOSUB 3400
 540 NEXT I
 550 GOSUB 1000
 560 GOSUB 6000
 570 PRINT C$;
 580 LET P = 2
 590 GOSUB 1000
 600 GOSUB 6000
 610 PRINT TAB(18) C$
 620 GOSUB 5000
 630 GOTO220
1000 REM  "HAND ANALYZER"
1010 FOR I = 1 TO 13
1020 LET H(I) = 0
1030 NEXT I
1040 FOR I = 1 TO 5
1050 GOSUB 1620
1060 LET H(J) = H(J)+1
1070 NEXT I
1080 FOR I = 1 TO 10
1090 LET R(I) = 0
1100 NEXT I
1110 FOR I = 1 TO 13
1120 LET J = H(I)
1130 IF J = 0 THEN 1250
1145 IF H(I) <> H(I-1) THEN 1160
1150 LET R(6) = R(6)+J
1160 IF J <> 2 THEN 1210
1170 IF R(9) = 0 THEN 1200
```

```
1180 LET R(8) = I
1190 GOTO 1210
1200 LET R(9) = I
1210 IF J <> 3 THEN 1230
1220 LET R(7) = I
1230 IF J <> 4 THEN 1250
1240 LET R(3) = I
1250 NEXT I
1260 IF R(6) = 5 THEN 1280
1270 LET R(6) = 0
1280 FOR I = 13 TO 1 STEP -1
1290 LET R(10) = I
1300 IF H(I) <> 0 THEN 1320
1310 NEXT I
1320 IF R(7) = 0 THEN 1350
1330 IF R(8) = 0 THEN 1350
1340 LET R(4) = R(7)
1350 FOR I = 1 TO 5
1360 GOSUB 1530
1370 IF J <> R(5) THEN 1440
1380 NEXT I
1390 LET R(5) = 6
1400 IF R(6) = 0 THEN 1440
1410 LET R(2) = R(10)
1420 IF R(10) <> 13 THEN 1440
1430 LET R(1) = 13
1440 IF P = 2 THEN 1490
1450 FOR I = 1 TO 10
1460 LET A2(I) = R(I)
1470 NEXT I
1480 RETURN
1490 FOR I = 1 TO 10
1500 LET B2(I) = R(I)
1510 NEXT I
1520 RETURN
1530 IF P = 2 THEN 1590
1540 LET R(5) = A1(1)
1550 LET J = A1(I)
1560 IF R(5) = J THEN 1580
1570 LET R(5) = 0
1580 RETURN
1590 LET R(5) = B1(1)
1600 LET J = B1(I)
```

```
1610 GOTO 1560
1620 IF P = 2 THEN 1650
1630 LET J = A(I)
1640 RETURN
1650 LET J = B(I)
1660 RETURN
1999 REM  "DRAW HANDS"
2000 GOSUB 8000
2010 FOR I = 1 TO 5
2020 GOSUB 8100
2030 LET A(I) = C
2040 LET A1(I) = S
2050 GOSUB 8100
2060 LET B(I) = C
2070 LET B1(I) = S
2080 NEXT I
2090 RETURN
2999 REM  "SET UP CARD PRINT"
3000 DATA 2,3,4,5,6,7,8,9,10
3010 DATA JACK,QUEEN,KING,ACE
3020 FOR I1 = 1 TO 13
3030 READ C$
3040 IF C = I1 THEN 3060
3050 NEXT I1
3060 RESTORE
3070 LET S$ = " HEARTS"
3080 IF S = 1 THEN 3140
3090 LET S$ = " CLUBS"
3100 IF S = 2 THEN 3140
3110 LET S$ = " DIAMONDS"
3120 IF S = 3 THEN 3140
3130 LET S$ = " SPADES"
3140 RETURN
3399 REM  "CARD PRINTER"
3400 LET C = A(I)
3410 LET S = A1(I)
3420 GOSUB 3000
3430 PRINT C$; S$;
3440 LET C = B(I)
3450 LET S = B1(I)
3460 GOSUB 3000
3470 PRINT TAB(18) C$; S$
3480 RETURN
```

```
4999 REM  "PICK THE WINNER"
5000 FOR I = 1 TO 9
5010 LET A2(10) = A2(10)+A2(I)
5020 LET B2(10) = B2(10)+B2(I)
5030 NEXT I
5040 FOR I = 1 TO 10
5050 IF A2(I) + B2(I) <> 0 THEN 5070
5060 NEXT I
5065 GOTO 5190
5070 IF A2(I) = B2(I) THEN 5160
5080 IF A2(I) > B2(I) THEN 5130
5090 PRINT TAB(9) "I";
5100 LET M = M+K
5110 PRINT " WIN $"K/2
5115 PRINT "YOU = $"Y"       ME = $"M
5120 RETURN
5130 PRINT TAB(9) "YOU";
5140 LET Y = Y+K
5150 GOTO 5110
5160 LET A2(I) = 0
5170 LET B2(I) = 0
5172 LET A2(10) = 0
5174 LET B2(10) = 0
5180 GOTO 5000
5190 PRINT "TIE HAND"
5200 LET M = M+K/2
5210 LET Y = Y+K/2
5220 GOTO5115
5999 REM  "PRINT HAND TYPE"
6000 DATA ROYAL-FLUSH,STRAIGHT-FLUSH
6010 DATA 4-OF-KIND,FULL-HOUSE,FLUSH
6020 DATA STRAIGHT,TRIPS,2-PAIRS
6030 DATA PAIR,HIGH-CARD
6040 FOR I = 1 TO 13
6050 READ Q$
6060 NEXT I
6070 FOR I = 1 TO 10
6080 READ C$
6090 IF R(I) <> 0 THEN 6110
6100 NEXT I
6110 RESTORE
6120 RETURN
6999 REM  "ADD SCORE"
```

```
7000 LET M = M-Q
7010 LET Y = Y-Q
7020 LET K = Q+Q
7030 LET L = 20
7040 IF M < L THEN 7050
7044 IF Y > L THEN 7060
7046 LET L = Y
7048 GOTO 7060
7050 LET L = M
7055 IF Y < M THEN 7046
7060 LET Q = 0
7070 RETURN
7999 REM  "GENERATE DECK"
8000 FOR I = 1 TO 52
8010 LET D(I) = I
8020 NEXT I
8030 RETURN
8099 REM  "DRAW CARDS"
8100 LET C = INT(100*RND(1))
8110 IF C < 1 THEN 8100
8120 IF C > 52 THEN 8100
8130 IF D(C) = 0 THEN 8100
8140 LET D(C) = 0
8150 LET S = 1
8160 IF C > 13 THEN 8180
8170 RETURN
8180 LET C = C-13
8190 LET S = S+1
8200 GOTO 8160
9000 REM  "RANDOM NUMBER ROUTINE"
9010 LET Z = RND(1)
9020 RETURN
```

G

Gunners

Gunners is simple and saucy, quick to code and fun to play, especially for those just breaking the two digit age barrier.

The game of *Gunners* was inspired by that commercialized plastic board-and-peg game called *Battleship*, which was itself no doubt inspired by that pencil-and-paper game of the same name. Instead of each player having to keep track of all of those little boats, though, each gunner has only one weapon—a tank. The idea of ten-by-ten grids is retained, and the players trade shots in an effort to blow away each other's tank. Thus the similarity to the game played with warships.

There are other differences. In *Gunners* the computer picks the grid locations for each contestant and provides spotter reports after each shot is fired. Printed out advice such as *"low left"* or *"high right"* is meant to help the gunner to lob successive rounds onto his opponent.

The phantom spotter's vocabulary is limited to just four clues: *high, low, left,* and *right*. The report may include only one of these, or it may be of the form mentioned above, using a pair of words. Because the grids are formed as intersecting columns and rows, once the clues fall into a one-word format all following shots that proceed along the indicated column or row are bound to score.

Lady Luck does pose a hazard to either player, but fortunately with equal odds for both. It is possible to score a direct hit with a single shot, and it is equally possible to home in on the correct column or row at the outset. Over the long run, however, the

players with the best strategies are likely to win more often than they lose. We will leave it to the eventual players of *Gunners* to determine what the best firing patterns are. Our purpose here is to help you get the game up and running.

BUILDING GUNNERS

In essence this is just another number-guessing game. It does differ from most in that there are two hidden numbers—one for each of two human players; and they are competing to see who can guess their opponent's number first.

There is a *column* and *row* connotation for each of the concealed locations, also. These could be generated internally as four separate integers—two for each player—but the technique used in

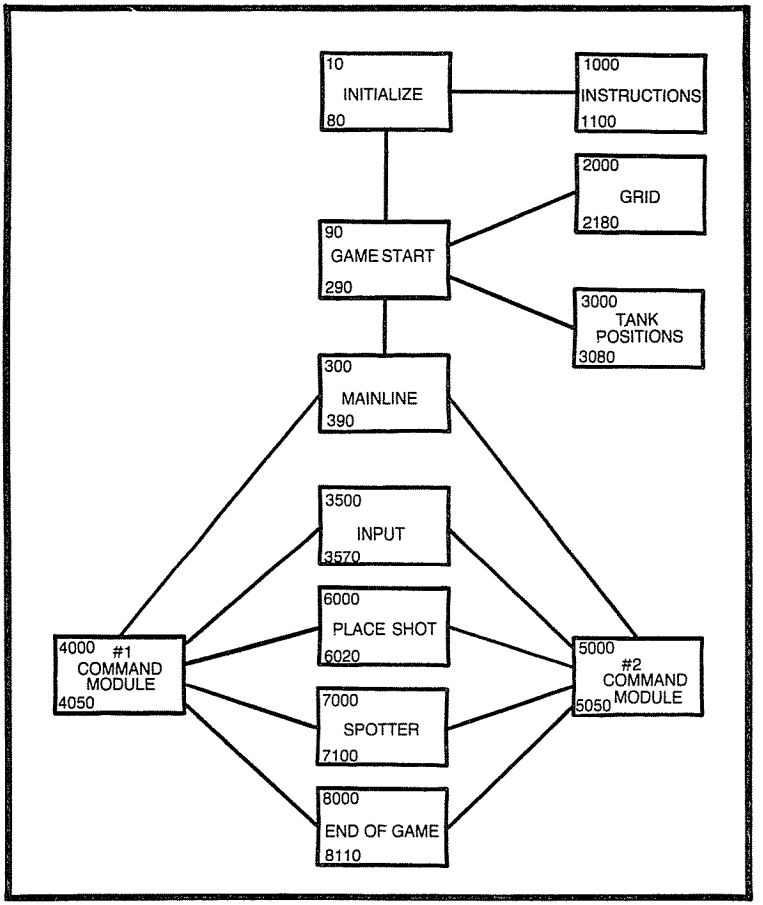


Fig. G-1. Program template for Gunners.

this program splits apart a two-digit whole number to derive the column-and-row value. Actually, there doesn't seem to be any special reason to prefer one method over the other, since either would require about the same amount of code and execution time. The choice here is simply my usual one.

Notice that the players in this game are supposed to enter both a column and a row number. These inputs are compared to those of the computer's, which are first generated as a pair of two-digit numbers (one for each tank's location). This whole scheme can be contrasted with that of *Abstract*. There, the player inputs a whole number which is parsed into individual digits for comparison to RND-generated digits. Incidental, but academically interesting.

There are several other tasks in *Gunners*, more than just that of the numbers. Figure G-1 illustrates all of the tasks and their conceptual placement. This program template also provides an alternative design method for controlling player sequencing.

All game programs that provide for multiple players must have some mechanism for knowing whose turn it is. Typically a player number is used for this. During a cycle of play the number is held constant, and routines, such as for player-dialog, scorekeeping and victory determination, use that constant. Between rounds, then, the player number is switched by arithmetic or some other means. A player number is used in *Gunners* also, but it serves only one purpose.

During a loop through the mainline the player number is checked for content and a subroutine jump is taken to one of two command modules. The two modules are alike, totally autonomous while in command. Yet they both depend upon a common set of supporting routines. All instructions that are sensitive to player identification are contained within the appropriate command module. And both modules contain identically coded subroutine jumps to the commonly used modules.

There is one other feature of *Gunners* that is slightly unusual. The program knows the players' names. Because a separate subprogram is used by the player in command it is easy to announce whose turn it is in a personal manner. Kids love this feature—the computer can remember their names. And there is even a bit of humor attempted by the startup sequence that accepts their self-introductions.

The entire business of the numbers, the names, command modules, and all of the rest are revealed in the following study on the guts of *Gunners*.

GUNNERS' INTERNALS

Although the number of statements in this program listing is over a hundred most of the coding is relatively simple. The program initialization sequence is just like most others in this book, and it is followed by a game startup sequence. It is there that we encounter the first of the several traits that characterize *Gunners*.

Out of line 100 the players are asked whether they would like to be provided with a printed grid. This, like the game's description, is an optionally invoked routine. Unlike the description process this option is afforded at the beginning of each running of the game. This replay option is possible because the mainline entry point for another game is at statement 90.

The column-and-row picture that is provided consists of lines of asterisks, with appropriate column numbers across the top and row numbers down the side. A brief description is in order at this point as to how this is done by the module located between statements 2000 and 2180.

A facsimile of the grid that can optionally be printed is shown in Fig. G-2. The print module begins by printing the header (COLUMNS) and the series of numbers one through ten. A blank line

```
                    COLUMNS
    1  2  3  4  5  6  7  8  9  10
    *  *  *  *  *  *  *  *  *  *  1
    *  *  *  *  *  *  *  *  *  *  2
    *  *  *  *  *  *  *  *  *  *  3
R
O   *  *  *  *  *  *  *  *  *  *  4
W
S   *  *  *  *  *  *  *  *  *  *  5
    *  *  *  *  *  *  *  *  *  *  6
    *  *  *  *  *  *  *  *  *  *  7
    *  *  *  *  *  *  *  *  *  *  8
    *  *  *  *  *  *  *  *  *  *  9
    *  *  *  *  *  *  *  *  *  *  10
```

Fig. G-2. The Gunners grid.

occurs then in 2060, the alpha variable A$ is loaded with a string of alternating asterisks and spaces. Each row of the matrix is output by simply printing the A$ value followed by the row number.

Since it was desired to include the ROWS caption in a vertical form along the side the grid is printed in three sections. The first three lines of asterisks are done by the FOR-NEXT loop in statements 2070, 2080, and 2090. Rows 4 through 7 are output by discrete expressions, and each includes one of the letters *R, O, W, S,* in that order. The print instructions in these four lines (2100 through 2130) are easily constructed using the A$ constant, with the appropriate trailing number from the series four through seven. The third and final step prints the remaining three rows in the same manner that the first three were done; this time by lines 2140, 2150, and 2160. The RETURN in 2180 relinks from this module back to the game's startup area at line 140.

From the relink point, or from statement 120 if the grid option is bypassed, the program unconditionally branches to line 200. It is here that the players are introduced to the computer.

Statements 200 through 240 ask for two names to be typed, one at a time; they are accepted into storage at P1$ and P2$. Lines 250 and 260 ask whether the names, as typed, are satisfactory. If the person doing the typing responds with N (for no), the whole introduction sequence is repeated. Otherwise, a couple of simple tests are made on the contents of P1$ and P2$. There is no logical need for these tests—they just add to the fun.

The conditionals in lines 272 and 274 do a quasi-check for alphabetical content in the name fields. Effectively, only the first letter is looked at, and only to the extent that the names must begin with A or some higher code. This at least precludes blanks and numbers being entered as nick-names. The comparison in line 276 goes one step further: identical aliases are rejected. This check is necessary; otherwise there would be no way for the computer to know who's shooting at whom.

A failure of any of these tests results in a bit of dialog that tends to show how smart the computer is. When the computer asks:

"NAMES OK (Y OR N)"

If the player answers Y but the tests fail the retort is:

"NO THEY'RE NOT!"

The program's exhortation is printed by statement 150, which is gotten to by whichever condition is disallowed. Sequential

fallthrough occurs from line 150 back into the introduction exercise. When the two entered names are finally different and they consist of at least one letter, the game gets under way.

From statement 280 a subroutine jump is taken to the module that dispassionately hides the tanks in their respective grids locations. Since this is done only once during the game's initialization this is an appropriate point to study the mechanics involved.

Deploying the Tanks

Only two expressions are used in *Gunners* to defilade the tank's combat positions (that's army talk for picking their parking spot). Lines 3000 and 3020 are both alike; they each generate a random number in the range 00 to 99. The first goes into P1 and the second in P2. By now you have probably surmised that the symbol P is consistently used to mean player.

A scan of the module's length from line 3000 to 3080 shows the RND expressions and an obvious gap in the line numbering following each. The gaps are intentional. If any difficulty is had in getting the kinks out of this program the following two statements will help immeasurably.

```
3010 PRINT P1
• • • •
3030 PRINT P2
```

While we are here two other housekeeping chores are taken care of: establishing who gets the first shot, and the starting of a shot counter with one. The *S* variable is always initialized by statement 3060, but 3050 is normally only accessed once. When *Gunners* is first loaded, P is bound to have a zero in it. Most systems initialize a newly loaded program with zip in all numeric variables. In any event if the program is restarted from the top, *P* is forced to zero by line 90 during program initialization.

The purpose of the conditional branch in line 3040 is to maintain the integrity of the alternation of the players' turns. By continuing the scheme of taking turns right on through from one game to the next, the capricious odds for lucky shots are somewhat nullified.

The RETURN statement at the bottom of this module lets the program continue at line 290, where it is announced which player gets to shoot first. From there on, beginning with line 300, the program is well into the mainline loop. Reentry into the mainline is either here or at line 320, depending on who fired last. Each time the first player's name comes up the shot counter is displayed by the print statement in line 300.

The conditional expression in line 320 checks whether or not P is a one, and either a branch to line 360 occurs or player 2 is up and the sequence from line 330 on is entered into. Both of the remaining sequences are comparable. The tank commander's name is displayed, P is surreptitiously flipped, and a jump to the appropriate command module is made. (Note the GOSUB in lines 350 and 380.) The subroutine return from either is followed by a check whether the game was ended while that player was in command. If either zaps the other a GOTO 100 restarts the game; otherwise the mainline is cycled through to permit a retaliatory shot.

Command Module

We will now look at one of the command modules. Because they are essentially the same only one must be studied. The twin routines are in lines 4000 through 4050 and 5000 through 5050. Let's look at the first, which starts in line 4000.

Right off of the bat a jump to line 3500 is taken. There the gunner is asked for fire-control directions. If the C and R values accepted by line 3506 are valid, a one is subtracted from both and a RETURN is effected. Either way, a return to the command module isn't allowed until the column and row workers each have a single digit in the zero to ninth range. When the relink is made an immediate test is executed to see whether a hit was scored.

The expression in line 4010 (and line 5010) is mechanically simple. Whatever is in C is multiplied by ten and the R value is added to it. Combined in this manner, the resulting two-digit number is structurally the same as the 00 to 99 value that is being maintained in P1 (and P2). In the event the CR value is an exact match with the opponent's tank the game is over. A branch from the command module to line 8000 (or 8090) announces the winner. The end-of-game module halts with an option to replay. If taken the Y response is carried along, following the RETURN (in line 7100), and the command module allows another game to be set up. If anything other than Y is typed the program simply terminates.

About the spotter and the clues: if the match is not exact when tested near the entry point into either command module, two more subroutines are called. The first of these, a GOSUB 6000, follows a simple move command to load T (temporary variable) with the opponent's grid location.

The module that is defined from line 6000 to 6020 parses the combined P1 (or P2) value into two digits. Statements that do this are in lines 6000 and 6010, working on the temporary copy of P1 that is in T. The result is placed into X and Y, corresponding with C and R

(the column and row workers). With the numbers now structured alike and aligned, the RETURN in 6020 is taken; so it's back to the command module for another jump, now to line 7000.

The spotter's report is done from line 7000 to 7100. Here X and C are compared, and so are Y and R. As the program bumps down through this sequence, each test results either in an immediate PRINT or a bypass to the next conditional. Because of the presentation sequence, not more than two clues can result, and they cannot be logically contradictory.

No doubt General Patton would approve if he could have seen *Gunners*.

THE PROGRAM

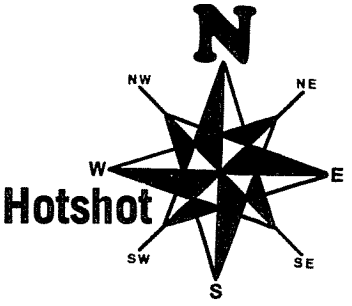
```
 10 REM  "GUNNERS"
 20 REM
 30 GOSUB 9000
 40 PRINT "WANT INSTRUCTIONS (Y OR N)";
 50 INPUT Q$
 60 IF Q$ = "N" THEN 90
 70 PRINT
 80 GOSUB 1000
 90 LET P = 0
100 PRINT "WANT GUNNER'S GRID (Y OR N)";
110 INPUT Q$
120 IF Q$ = "N" THEN 200
130 GOSUB 2000
140 GOTO 200
150 PRINT "NO THEY'RE NOT!"
200 PRINT "TYPE 2 PLAYER'S NAMES"
210 PRINT "GUNNER #1  ";
220 INPUT P1$
230 PRINT "GUNNER #2  ";
240 INPUT P2$
250 PRINT "NAMES OK (Y OR N)";
260 INPUT Q$
270 IF Q$ = "N" THEN 200
272 IF P1$ < "A" THEN 150
274 IF P2$ < "A" THEN 150
276 IF P1$ = P2$ THEN 150
280 GOSUB 3000
290 PRINT "#"P" GETS THE FIRST SHOT"
300 PRINT "ROUND #"S
320 IF P = 1 THEN 360
```

```
 330 PRINT P2$;
 340 LET P = 1
 350 GOSUB 4000
 352 LET S = S+1
 354 IF Q$ = "Y" THEN 100
 356 GOTO 300
 360 PRINT P1$;
 370 LET P = 2
 380 GOSUB 5000
 385 IF Q$ = "Y" THEN 100
 390 GOTO 320
 999 REM  "DESCRIPTION"
1000 PRINT "TWO PLAYERS SHOOT FROM A TANK AT"
1010 PRINT "EACH OTHER OVER A HILL."
1020 PRINT "(THEY CAN'T SEE EACH OTHER.)"
1030 PRINT "I'LL ACT AS SPOTTER FOR BOTH."
1040 PRINT "INPUT IS A ROW/COLUMN VALUE"
1045 PRINT "LIKE THIS:"
1050 PRINT "   AIM (R,C)?  3,4"
1060 PRINT "THE GRID IS 10 X 10, SO NEITHER"
1070 PRINT "R NOR C MAY BE OVER 10 OR LESS"
1080 PRINT "THAN 1."
1090 PRINT
1100 RETURN
1999 REM  "GUNNER'S GRID"
2000 PRINT TAB(9) "COLUMNS"
2020 PRINT TAB(3) "1 2 3 4 5 6 7 8 9 10"
2050 PRINT
2060 LET A$ = "* * * * * * * * * * "
2070 FOR I = 1 TO 3
2080 PRINT TAB(3) A$;I
2090 NEXT I
2100 PRINT "R  " A$ " 4"
2110 PRINT "O  " A$ " 5"
2120 PRINT "W  " A$ " 6"
2130 PRINT "S  " A$ " 7"
2140 FOR I = 1 TO 3
2150 PRINT TAB(3) A$; I+7
2160 NEXT I
2170 PRINT
2180 RETURN
2999 REM  "RANDOM GRID LOCATIONS"
3000 LET P1 = INT(100*RND(1))
```

```
3020 LET P2 = INT(100*RND(1))
3040 IF P <> 0 THEN 3060
3050 LET P = 1
3060 LET S = 1
3070 LET Q$ = "X"
3080 RETURN
3499 REM  "PLAYER INPUT"
3500 PRINT "  AIM (R,C)";
3502 LET C = 0
3504 LET R = 0
3506 INPUT C,R
3510 IF C < 1 THEN 3560
3520 IF C > 10 THEN 3560
3530 IF R < 1 THEN 3560
3540 IF R > 10 THEN 3560
3542 LET C = C-1
3544 LET R = R-1
3550 RETURN
3560 PRINT "ILLEGAL..."
3570 GOTO 3500
3999 REM  "QUALIFY #1'S SHOT"
4000 GOSUB 3500
4010 IF C*10+R = P2 THEN 8000
4020 LET T = P2
4030 GOSUB 6000
4040 GOSUB 7000
4050 RETURN
4999 REM  "QUALIFY #2'S SHOT"
5000 GOSUB 3500
5010 IF C*10+R = P1 THEN 8090
5020 LET T = P1
5030 GOSUB 6000
5040 GOSUB 7000
5050 RETURN
5999 REM  "PLACE SHOT"
6000 LET X = INT(T/10)
6010 LET Y = T-INT(X*10)
6020 RETURN
6999 REM  "SPOTTER'S REPORT"
7000 PRINT "MISSED:  ";
7010 IF C =< X THEN 7030
7020 PRINT "LOW ";
7030 IF C => X THEN 7050
```

```
7040 PRINT "HIGH ";
7050 IF R =< Y THEN 7070
7060 PRINT "RIGHT";
7070 IF R => Y THEN 7090
7080 PRINT "LEFT";
7090 PRINT
7100 RETURN
7999 REM  "WRAP IT UP"
8000 PRINT P1$; " IS HIT"
8010 PRINT P2$;
8020 PRINT " IS THE BEST GUNNER."
8030 PRINT
8040 PRINT "ANOTHER GAME (Y OR N)";
8050 INPUT Q$
8060 IF Q$ = "Y" THEN 7100
8070 PRINT "THE END"
8080 END
8090 PRINT P2$; " IS HIT"
8100 PRINT P1$;
8110 GOTO 8020
9000 REM  "RANDOM NUMBER ROUTINE"
9010 LET Z = RND(1)
9020 RETURN
```

H



In the previous chapter Gunners permitted humans to shoot at each other. The role of the computer there was as a dispassionate mechanic. In this game human aficionados try to zap the elusive machine.

Once again, the foe is ensconced in a grid, ten numbers to the side, and to shoot is to enter column and row coordinates. Meanwhile, inside of the program, the trail of your shots is watched, and the computer tries to escape your shot pattern. After each round is fired you are tersely advised as to the accuracy of your shot, and the computer may or may not take evasive action. In either event the computer can move only one square per turn, in any direction it chooses.

A particular meaning applies to each of the three spotter clues. A *hot shot* is one that has landed on an adjacent square (above, below, to either side, or on either adjacent diagonal) to the computer's entrenched position. When you make a *hot shot* the computer has to move.

Advice that you were *close* should be interpreted similarly, the difference being that your round landed two squares distant. That is, there is one square between the target and the strike of your shot. If you were close, the computer may move one square in any direction, including on the diagonals. Then again, it may not make a move; the choice is the computer's.

If the computer advises that you *missed*, it is not permitted to move. To have *missed* simply means that your shot was beyond the

definition of the other two clues (and you certainly did not score; otherwise the game would be over).

Sound like fun? It is. Even after you have mastered the intricacies of how the program calculates its moves it can still be exasperating, sometimes, to defeat what is in principle a simple strategy.

Every move made by the computer is based on a finite scheme which depends on a primitive implementation of artificial intelligence. That is, it looks at a brief history of your shots in an attempt to discern whether you are proceeding according to some plan. To do so, the program has to "see" the grid and the relative placement of your shots. What the program sees is looked at in the following manner.

THE VIRTUAL LOGICAL GRID

Players of *Hotshot* and the program itself depend on a ten by ten virtual grid (it exists in concept only). The one shown in Fig. H-1 assists in seeing how the program works, but oddly enough no such array really exists; at least it isn't defined as a table structure. All of the program's processes are constructed in such a way that there is a virtual *logical grid.*

The player thinks of the grid as being numbered from one to ten across the top and one to ten down the side. When a shot is entered the program first subtracts a one from both the column and row number; then it combines the results to make a whole number between zero and ninety-nine, inclusive.

A simple bit of arithmetic can derive a vector from any two placement values. The difference between two numbers, as shown in the drawing, indicates whether the two locations are on a horizontal, vertical, or a diagonal line. The sign of the result after subtraction indicates direction.

If a first shot is made on square 33, and the next is on 34, subtraction will give a positive 1. The implication is that the shots are moving along a vertical line from top to bottom. By orienting the grid like a map, with the top to the north, the sequence of 33, 34, 35, indicates a due-south vector along the logical column 3. In the same way, if 34 is subtracted from 33, the absolute difference of one still means a vertical track, but the inverted sign (minus) indicates a direction of north.

Notice, also, that distance along a line can be determined from even multiples. For example, 54 from 74 is +20. If a player first shoots at 54, then 74, he might be tracking due-east, shooting at every other square. Then again, he might not pick 94 next. Who

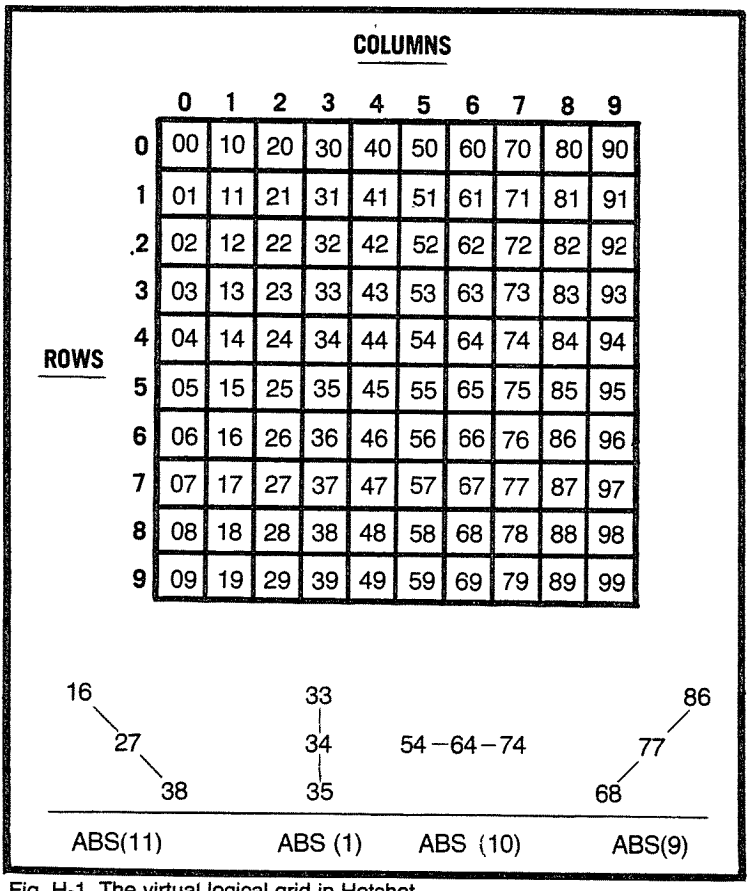


Fig. H-1. The virtual logical grid in Hotshot.

knows what some people are apt to do? Yet, it is possible that the player is using some sort of pattern to locate the target.

The logic of this program presumes the player may be tracking along some line, in a given direction. When a move has to be made, it is best not to make a move that coincides with such an obvious trail, just in case.

THE PROGRAMMED STRATEGY

Assume that the player has been shooting at alternate squares along a given line. If a *close* clue is given, the player may think he has the target pegged at two squares farther along. (The target might also be at a right angle to the vector he is on.) If the program knows that the track is straight it attempts to move one square ahead to foil any such straddling tactic.

The player has no assurance that the program will attempt to evade him, though, when he is only close. The computer's decision is based simply on whether a programmed move would encounter the grid's boundary. If the computed move wouldn't be legal the program simply defaults and no move is made. The inference here is that the player may crowd the target into the side of the grid and zap it—if he is tracking true, and if the computer indicated a *close* when his last strike was three squares from the edge.

For a *hot shot*—one square off the mark—the computer uses a different tactic. In this case the player knows the computer has to move. To make it more difficult to anticipate the move, instead of staying on the same vector the program forces a move at a right angle, either right or left of the indicated track. The direction change is programmed in, but the boundary problem will cause it to reverse its usual choice whenever an illegal move is attempted. Here again is an inference for the player. Work the target into a corner, but remember the prior maneuvers: the program's probable move can be predicted if you know where it last was.

The history file on which the program does its analysis is brief: only two shots are looked at—this one and the last one. And all of the forced moves are made, even if there is no straight line indicated by comparing the two most recent shots. In the event their difference is not one, nine, ten, or eleven (or double any of these) the program works as if the difference is one. The sign of the actual difference is used, however, to determine whether an east or west turn is taken following a randomly placed *hot shot.*

The program for playing *Hotshot* is not very long; yet there is a certain intricacy in its internal logic. By modularizing its design the separate tasks can be studied individually, thus simplifying what could otherwise be a complex program.

PUTTING HOTSHOT TOGETHER

The program's template is shown in Fig. H-2. The first three blocks at the upper left are easily understood. The program's initialization sequence includes the option to print the game's instructions. There is also, like Gunners, an option to print out a grid from the start-game area. The third block in the sequence is the one executed just before each game gets under way, so it includes a jump to the subroutine that initially hides the target. Then, the mainline.

The dispatching sequence out of the mainline is, in order, to lines 4000, 5000, and 6800. The first jump is made to get the player's input, and the second is to output the appropriate clue. The

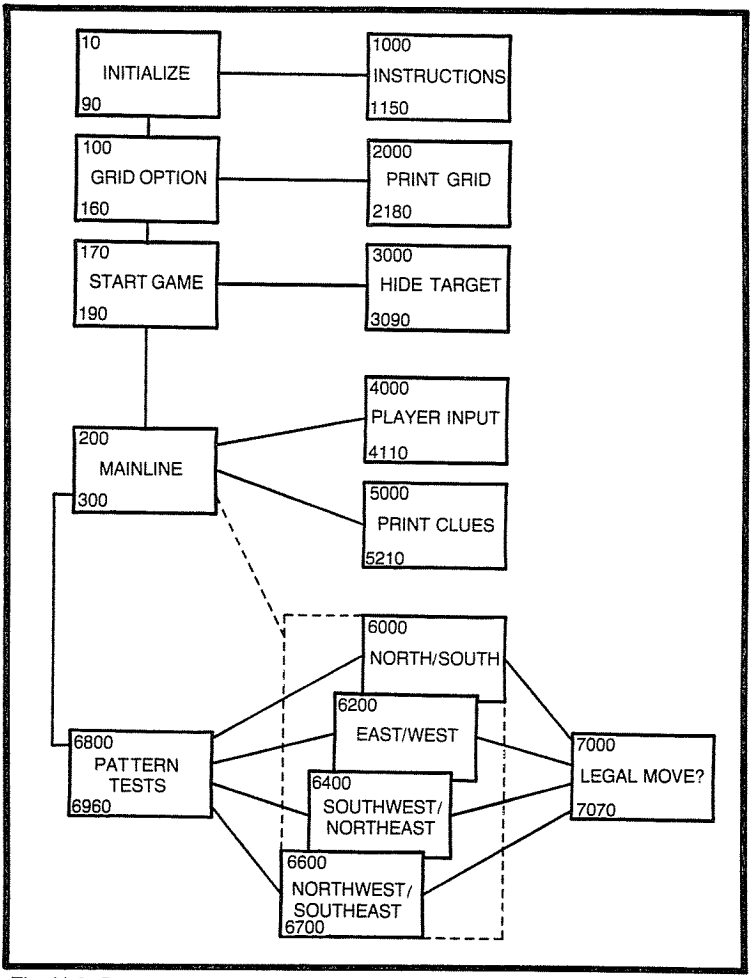


Fig. H-2. Program template for Hotshot.

third jump, to 6800, is to the crux of the program, to figure out what to do, based on what the player did. After the last two shots are compared one of the other 6000-series blocks is branched to if a move is required.

The dotted lines in Fig. H-2 show the RETURN path back to the mainline. All of this area commonly shares the subroutine which begins at line 7000 to qualify the legality of computed moves, and the RETURN from there is always back to the calling point. There are certain inter-relationships between the blocks that are surrounded by the dotted line. What those relationships are will become evident when it is seen how the program actually works.

MAKING HOTSHOT WORK

Down to line 180 in the program listing the logic is both routine and rote. The dialog from lines 40 through 100 includes the jump to print the game's instructions; that from line 110 through line 170 provides for the grid, printed at the player's option. It is out of line 180 that the game really begins, with a housekeeping jump to line 3000.

The RND expression in line 3000 generates a two-digit integer in the 00 to 99 range. From line 3010 to 3040 the intent is to enforce the random generation function to give an initial target location at least one square removed from the grid's outside boundaries. Needless to say the player is not told anywhere that the target is never initially near an edge. After all the machine is at an intelligence disadvantage—in most cases anyway.

The player's shots are accepted between lines 4000 and 4110. Notice that as this subroutine begins the round number (*R*) is incremented by one. It was initialized at zero during housekeeping; so from here on each time this routine is called the prompt line will show the number of the shot then being input.

The player's row and column numbers are accepted into the *X* and *Y* variables in line 4020. The next two expressions insure that no decimal junk nor any negative signs are allowed to clutter up these two workers. Because both the *X* and *Y* values are immediately reduced by one the range tests in lines 4050 and 4060 only have to check for an upper limit of nine. The expression in line 4070 combines the two input values into the field known as *S* (for shot). If the result is usable, the RETURN in line 4080 goes back to the mainline.

For any input that is illogical, an *Invalid shot* message is printed by line 4090, and control is returned to 4010 to force another entry. The exception: an entry of ninety-nine as a row value will condition the programmer's prerogative to conclude the game without actually winning.

The next module that is called upon extends from lines 5000 to 5210. It is here that the player's shot is spotted, the appropriate clue is printed, and the *M* worker (*missed*) is conditioned. If the comparison in line 5000 finds that the game should be ended, *M* is returned to the mainline with a zero in it. Otherwise, the game goes on.

As this subroutine is continued, *M* is set to one in line 5040. The expression in line 5050 generates the actual distance from the shot to the target (*L*), and places the signed result in *D*. In line 5060 this value is converted to an absolute integer in *A*. This completes the preparation for the clue-generating sequence that follows.

From lines 5070 to 5130 is a two-times FOR-NEXT loop that does comparisons on the absolute distance between the shot that was just made and the current location of the target. The four conditionals are testing for whether *A* is equal to a one, nine, ten, or 11. If so, a *hot shot* exit is taken to line 5170. If all four conditionals fail, however, *M* is increased to a two by line 5120, and the tests are tried again. By using *M* as a multiplier, *A* can be checked if it contains a two, eighteen, twenty, or twenty-two. If so, the same exit to line 5170 is taken, and there the *close* message is output.

Whenever the loop runs all the way through twice the player's shot is more than two squares away from the target; so *M* is loaded with a three, *missed* is printed, and the game continues back in the mainline.

If back there in line 220 *M* is anything other than a zero a jump is made to line 6800. Although a *missed* shot *M* equals a three may already be known at this point, it is still necessary to go to the next module to update the *two-shot* history file before proceeding with another input.

Logic of the sequence that runs from lines 6800 to 6960 is very much like the one just studied. This time, *P* is the variable that is used to know whether the two-times FOR-NEXT loop runs only once, twice partially, or twice through all of the way. The numbers that are being tested have nothing to do with the target this time; rather, it is the player's most recent two shots that are analyzed.

In line 6810 and 6820 *D* and *D1* are set up using subtraction. Where *S* is still used to hold the shot, *S1* holds the last previous shot. Subtracting twice, once each way, will cause a negative result in *D* or *D1*, and vice versa. It is these opposites that permit the program to change its mind when making right-angle moves.

In line 6830 the prior shot is overwritten with the one just made. This is done in anticipation of the next-time use of these processes. Line 6840 now tests for a missed shot. If such was the case the RETURN to the mainline is taken out of line 6960. Otherwise, something more arduous is required.

If the pattern tests learn that the absolute distance between the last two shots is one, nine, ten, or eleven the appropriate dispatch is made, and *P* holds a one to indicate the single-space situation. If the loop fails the first time through it is tried once more with a two in the multiplier. The same exits are possible on this lap, but they will only activate this time on two, eighteen, twenty, or twenty-two. If none of the above, *P* is made to be a three, and *M* is looked at in line 6940. This is done to decide whether the program has to make a move.

Recall that if M had a three in it the program doesn't get this far; Nor would it if M was zero. At this point only a one and a two are possible. If M was conditioned earlier with a two the last clue given was for a shot that was *close*. The option is easy: exit. The remaining choice is just as easy: the program has to make a move if M is equal to one. The branch that is taken is arbitrary, and it goes to the same module that would have been chosen if the player's pattern indicated a vertical track.

The four move modules are identical. Their only differences are the branching references used and the constants used to compute a move. The first of the four is labeled "north/south" (lines 6000 through 6100). The next is its *close* relative, "east/west" (lines 6200 through 6300). The other two move modules are similarly related, and they work in conjunction to affect moves on either diagonal (lines 6400 through 6500 and lines 6600 through 6700). Because they are all so much alike only one needs to be studied closely.

Looking at the one that starts in line 6000, notice the test for a one in P. If a one is the value in P the player's last two shots were single-stepping on the north/south axis. The program goes to the next module to do an east to west move in this case. The branch there is one line late; otherwise, an identical test would be encountered. It is these matching tests that will always cause a right-angle move for a patterned *hot shot* or for those moves that have to be made when no pattern is discernible.

If the pattern test does fail (or P has a two in it) the sign of D is looked at in line 6010 to determine direction. Depending on the outcome, the T-temporary worker is loaded with a modified location value in either line 6020 or line 6070. The jump to line 7000 is next, in either case, to see whether the computed move is a legal one.

The subroutine from line 7000 to line 7070 will leave T alone if it is usable, or 999 may be overlaid into T to show an attempted boundary violation. The other possibility is that the computed move is not a valid one, but the program doesn't have to make any move. That is why in line 7030 the M variable is checked for a two. The can't-go-straight-ahead option is set up by changing P to a nine (line 7060). Back to the calling module.

The first test there, in line 6035, checks the P=9 option. If so, on with the game. Otherwise, what does T have in it? If the test in line 6040 defaults the computed move is good: the location is updated with the tested move and the RETURN in line 6060 links back to the mainline.

If the request for a computed move comes back with a 999 the "other" *D* (being held in *D1*) is used, and line 6100 branches back to the top to make the program compute a right-angle move in the opposite direction. Simple, huh?

Notwithstanding the program's internal simplicity, the real hotshots in the family will be known by their ratios of close encounters and shots fired. Don't be surprised if someone other than yourself has better scores in the final line output at game's end.

THE PROGRAM

```
 10 REM  "HOTSHOT"
 20 REM
 30 GOSUB 9000
 40 PRINT "WANT DESCRIPTION (Y OR N)";
 50 INPUT Q$
 60 IF Q$ = "Y" THEN 90
 70 IF Q$ = "N" THEN 100
 80 GOTO 40
 90 GOSUB 1000
100 PRINT
110 PRINT "WANT GRID (Y OR N)";
120 INPUT Q$
130 IF Q$ = "Y" THEN 160
140 IF Q$ = "N" THEN 170
150 GOTO 110
160 GOSUB 2000
170 PRINT
180 GOSUB 3000
190 PRINT
200 GOSUB 4000
205 IF X = 98 THEN 250
210 GOSUB 5000
220 IF M = 0 THEN 245
230 GOSUB 6800
240 GOTO 200
245 PRINT "MISSED"R3"CLOSE"R2"HOT SHOTS"R1
250 PRINT "PLAY AGAIN (Y OR N)";
260 INPUT Q$
270 IF Q$ = "Y" THEN 100
280 IF Q$ <> "N" THEN 250
290 PRINT "SO LONG NOW ..."
300 END
1000 PRINT "I AM HIDING IN A 10 X 10 GRID."
```

```
1010 PRINT "YOU ARE SHOOTING AT ME."
1020 PRINT "AFTER EACH SHOT YOU'LL BE TOLD:"
1030 PRINT "  (1)  MISSED"
1040 PRINT "  (2)  CLOSE"
1050 PRINT "  (3)  HOT SHOT"
1060 PRINT "IF YOU 'MISSED' I WON'T MOVE."
1070 PRINT "IF YOU WERE 'CLOSE' I MAY MOVE."
1080 PRINT "A 'HOT SHOT' MAKES ME MOVE."
1090 PRINT "  (I CAN ONLY MOVE 1 SQUARE --"
1100 PRINT "  UP, DOWN, OR DIAGONALLY.)"
1110 PRINT "INPUT IS A ROW/COLUMN VALUE,"
1120 PRINT "LIKE THIS:  SHOOT (R,C)?  5,6"
1130 PRINT "READY";
1140 INPUT Q$
1150 RETURN
1999 REM  "PRINT GRID"
2000 PRINT TAB(9) "COLUMNS"
2020 PRINT TAB(3) "1 2 3 4 5 6 7 8 9 10"
2050 PRINT
2060 LET A$ = "* * * * * * * * * * "
2070 FOR I = 1 TO 3
2080 PRINT TAB(3) A$;I
2090 NEXT I
2100 PRINT "R  " A$ " 4"
2110 PRINT "O  " A$ " 5"
2120 PRINT "W  " A$ " 6"
2130 PRINT "S  " A$ " 7"
2140 FOR I = 1 TO 3
2150 PRINT TAB(3) A$; I+7
2160 NEXT I
2162 PRINT
2164 PRINT
2170 INPUT Q$
2180 RETURN
2999 REM  "HIDE TARGET"
3000 LET L = INT(100*RND(1))
3010 IF L < 11 THEN 3000
3020 IF L > 88 THEN 3000
3030 LET T = INT(L/10)
3040 IF T*10-L = 0 THEN 3000
3050 LET S1 = 0
3060 LET R = 0
3062 LET R1 = 0
```

```
3064 LET R2 = 0
3066 LET R3 = 0
3070 LET X = 0
3080 LET Y = 0
3090 RETURN
3999 REM  "PLAYER INPUT"
4000 LET R = R+1
4010 PRINT "("R")   SHOOT (R;C)";
4020 INPUT X,Y
4030 LET X = INT(ABS(X-1))
4040 LET Y = INT(ABS(Y-1))
4050 IF X > 9 THEN 4090
4060 IF Y > 9 THEN 4090
4070 LET S = Y+X*10
4080 RETURN
4090 PRINT "INVALID SHOT"
4100 IF X = 98 THEN 4080
4110 GOTO 4010
4999 REM  "PRINT CLUES"
5000 IF S <> L THEN 5040
5010 PRINT "ZAP -- GOT ME!"
5020 LET M = 0
5030 RETURN
5040 LET M = 1
5050 LET D = L-S
5060 LET A = ABS(D)
5070 FOR I = 1 TO 2
5080 IF A = M*1 THEN 5170
5090 IF A = M*9 THEN 5170
5100 IF A = M*10 THEN 5170
5110 IF A = M*11 THEN 5170
5120 LET M = 2
5130 NEXT I
5140 LET M = 3
5150 PRINT "MISSED"
5155 LET R3 = R3+1
5160 RETURN
5170 IF M = 2 THEN 5200
5180 PRINT "HOT SHOT"
5185 LET R1 = R1+1
5190 RETURN
5200 PRINT "CLOSE"
5205 LET R2 = R2+1
```

```
5210 RETURN
5999 REM "NORTH/SOUTH"
6000 IF P = 1 THEN 6210
6010 IF D < 0 THEN 6070
6020 LET T = L-1
6030 GOSUB 7000
6035 IF P = 9 THEN 6060
6040 IF T = 999 THEN 6090
6050 LET L = T
6060 RETURN
6070 LET T = L+1
6080 GOTO 6030
6090 LET D = D1
6100 GOTO 6000
6199 REM  "EAST/WEST"
6200 IF P = 1 THEN 6010
6210 IF D < 0 THEN 6270
6220 LET T = L-10
6230 GOSUB 7000
6235 IF P = 9 THEN 6260
6240 IF T = 999 THEN 6290
6250 LET L = T
6260 RETURN
6270 LET T = L+10
6280 GOTO 6230
6290 LET D = D1
6300 GOTO 6200
6399 REM  "SOUTHWEST/NORTHEAST"
6400 IF P = 1 THEN 6610
6410 IF D < 0 THEN 6470
6420 LET T = L-9
6430 GOSUB 7000
6435 IF P = 9 THEN 6460
6440 IF T = 999 THEN 6490
6450 LET L = T
6460 RETURN
6470 LET T = L+9
6480 GOTO 6430
6490 LET D = D1
6500 GOTO 6400
6599 REM  "NORTHWEST/SOUTHEAST"
6600 IF P = 1 THEN 6410
6610 IF D < 0 THEN 6670
```

```
6620 LET T = L-11
6630 GOSUB 7000
6635 IF P = 9 THEN 6660
6640 IF T = 999 THEN 6690
6650 LET L = T
6660 RETURN
6670 LET T = L+11
6680 GOTO 6630
6690 LET D = D1
6700 GOTO 6600
6799 REM  "PATTERN TESTS"
6800 LET P = 1
6802 IF S1 <> S THEN 6810
6804 LET S = S+1
6810 LET D = S1-S
6820 LET D1 = S-S1
6830 LET S1 = S
6840 IF M = 3 THEN 6960
6850 LET A = ABS(D)
6860 FOR I = 1 TO 2
6870 IF A = P*1 THEN 6000
6880 IF A = P*10 THEN 6200
6890 IF A = P*9 THEN 6400
6900 IF A = P*11 THEN 6600
6910 LET P = 2
6920 NEXT I
6930 LET P = 3
6940 IF M = 2 THEN 6960
6950 GOTO 6210
6960 RETURN
6999 REM  "LEGAL MOVE?"
7000 IF T < 0 THEN 7030
7010 IF T > 99 THEN 7030
7011 REM  "IF LOW-T IS 0, LOW-L CAN'T BE 9"
7012 REM  "IF LOW-T IS 9, LOW-L CAN'T BE 0"
7013 LET T9 = 0
7014 LET L9 = 0
7015 IF T = 0 THEN 7017
7016 LET T9 = INT(T/10)
7017 IF L = 0 THEN 7020
7018 LET L9 = INT(L/10)
7020 IF T - T9*10 = 0 THEN 7023
7021 IF T - T9*10 = 9 THEN 7025
```

```
7022 RETURN
7023 IF L - L9*10 = 9 THEN 7030
7024 RETURN
7025 IF L - L9*10 = 0 THEN 7030
7026 RETURN
7030 IF M = 2 THEN 7060
7040 LET T = 999
7050 RETURN
7060 LET P = 9
7070 RETURN
9000 REM  "RANDOM NUMBER ROUTINE"
9010 LET Z = RND(1)
9020 RETURN
```

Invert

Invert: to reverse in position, order, or relationship. Thank you, Mr. Webster. That is exactly how this game is played—with a string of nine numbers, one each from the set one through nine. Two players compete, starting out with matching copies of a jumbled string of numbers. The object is to see who can straighten his or hers out first, by rearranging them into their natural sequence of from one to nine.

The competition: you and the computer. As the game proceeds, with the two of you taking turns, your individually manipulated strings are reprinted to show the results of each move. There is only one rule. You may pick any of the individual digits in your string to invert. Then all of the numbers from the leftmost position, through the one you pick, will be reversed in their order of presentation.

You always get the first turn. Your simplest strategy would be to duplicate whatever move the computer makes. This does require that you properly anticipate the computer's first move, but that isn't difficult because the *Invert* program uses only one algorithm. For want of a better name I call it the two-step-absolute algorithm. It is shown in Fig. I-1. Yes, it is simple. But, lest you think this is all too simple or too absolute, please read on.

There can be corporeal benefits from having this program in your game library. You can do pseudoscientific studies on your friends and neighbors. But don't tell them about the computer's ironclad algorithm. Repeated play is assumed to be the norm, so the

program maintains win counters. It can be very interesting to see just how long it takes for man to ascend beyond the simple intelligence of the machine.

This is not to imply that I am trying to ameliorate with those who have an affinity for categorizing everyone into archtypal groupings. Nevertheless, there does seem to be a distinct tendency for the players of *Invert* to align themselves with one of the following.

- Some never seem to catch on. Their moves always appear to be haphazard; perhaps depending on blind luck, or maybe they are cowered into believing that there is some mystique involved
- Many players soon detect the rhythm of the machine's moves. A few of these remain perplexed on their first move, indicating an intuitive grasp, yet they remain incapable of practicing what they know
- Probably the most amusing of all are those who accurately perceive how it is done, but that fail to identify the incompleteness of the two-step-absolute algorithm

As a programmer, and as you study the internals of *Invert*, you are bound to notice that the algorithm used here never takes advantage of any preexisting natural sequences imbedded within the randomly generated startup strings. Human perception can prevail over this failure, but only in those instances that some natural order is inherent when the strings are first generated.

There is also that purely humanistic trait: mistakes. The machine is not supposed to ever be guilty of carelessness. It will unerringly do its thing, time and again. If you are running neck-and-neck you should always win—unless you make a mistake. You are also enforced to use a two-step method, so it takes two rights to correct a wrong. This usually means that you will have lost the advantage that is inherent to having the first move. Because all human-type players are capable of an occassionally hasty move the program does permit a graceful out.

At the point in the play, when it's your turn, a pick of zero will trap out and ask if you would like to forfeit the game. If the zero entry was itself a careless keystroke you may advise the program with a no answer and be permitted to continue playing. On the other hand, if you can readily see that you are beaten, a yes response to the forfeit question will invoke a simple *"thank you"* from the computer and a new game may be started immediately with no further ado.

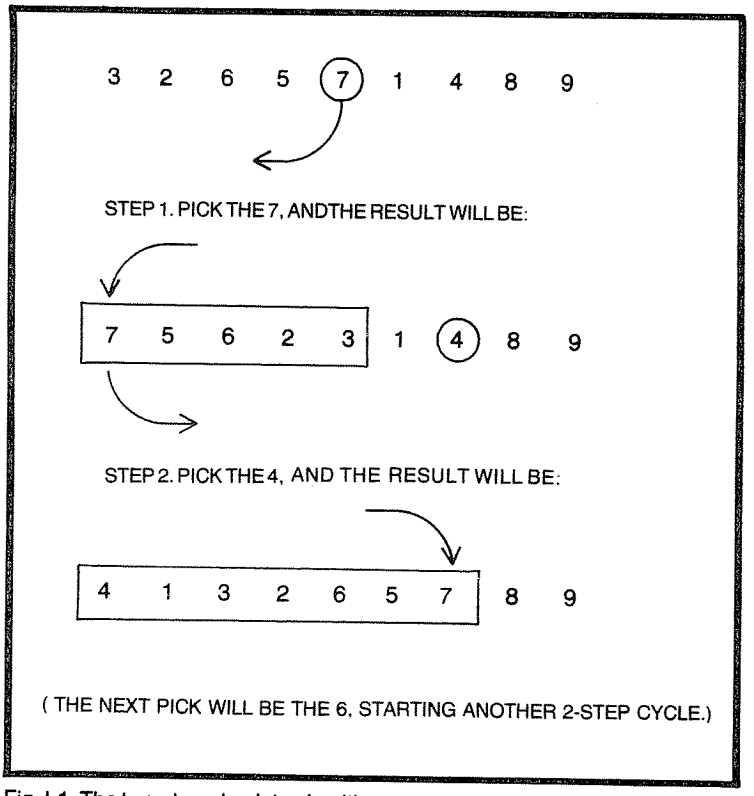

3 2 6 5 7 1 4 8 9
STEP 1. PICK THE 7, AND THE RESULT WILL BE:
7 5 6 2 3 1 4 8 9
STEP 2. PICK THE 4, AND THE RESULT WILL BE:
4 1 3 2 6 5 7 8 9
(THE NEXT PICK WILL BE THE 6, STARTING ANOTHER 2-STEP CYCLE.)

Fig. I-1. The two-step-absolute algorithm to rotate the next desired digit through the leftmost position.

In a nutshell, that's all there is to the game of *Invert*. The program for invert is worthy of study, though. As can be surmised the string of numbers is kept in a table; so there is at least the business of the topsy-turvy table gyrations—for both the human and the machine.

PROGRAM OVERVIEW

Here again the modularity theme prevails. The illustration in Fig. I-2 shows this as well as the manner in which the program's major tasks are laid out. This template does look much the same as others in this book for games that involve two players.

In this game one of the players is the computer, so there is a separate module to support the machine's turn. The player's move uses two modules, one for input and a separate one for manipulating his string. All of these supporting tasks are callable subroutines, with the mainline dictating their accessing sequence.

The mainline looping sequence begins with a jump to a string print module that outputs both player's strings, one above the other. The top one is labeled YOURS and the second one, MINE. The mainline flow then calls the player input module. This module outputs an operator prompt line that looks like this.

INVERT *n*-?

In lieu of *n* the program automatically inserts whatever the player's leftmost string number is to further insure that the player is studying his own string. The input module does quite a bit of checking to make sure that whatever is typed is a valid number—but not necessarily whether there is any rhyme or reason to the choice.

Upon return to the mainline, the invert string is called next. This module reverses that portion of the player's string that is delimited by the two numbers that appear in the INVERT *n* prompt. The numbers are only rearranged in memory at this point; printing comes later, after the computer has its turn.

Only one subroutine is needed for the computer to have a go. Logic of this module is such that the choice is in accordance with the two-step-absolute method described earlier. Manipulation of the MINE string is accomplished here, immediately, with nothing being printed yet. Printing of both strings is delayed until the start of another cycle, their respective contents being at this time whatever the results of each player's move was during this round.

When the mainline is returned to from the *computer's turn* the last major module is called up. This routine is constructed to test for a win and either allow the mainline to be recycled or close up the game with an option to start another.

There is a scorekeeping service provided by this module also. There are two counters maintained here—one for each player. Just before the replay option is allowed both scores are printed. If the counters are equal and the player elects to quit the program says THANKS FOR THE GAME, and program execution stops.

On the other hand when the exit route is chosen two other messages are possible. If the player's score is the higher of the two the program prints: GOOD-YOU'RE TOO CLEVER ANYWAY. But if the player hasn't yet caught up to the computer's score, a parting shot is made: CHICKEN.

Invert is meant to amuse, so there is quite a lot of this sort of dialog scattered throughout the coding; yet conceptually there is not much complexity in the program's mechanics.

Basically, operation of this program depends on the management of tables. There are three of these, and each are nine elements

deep. *Table A* and *table B* are used to house the human's and the machine's number strings. The third table (*C* for collector) is a work area, and it is alternately shared by the two string-manipulating processes.

When a game first begins, near the top of the mainline, a stand-alone task is called to generate the matching scrambled strings. During this task only *table A* and *table B* are used. The randomly sequenced numbers are shuffled in *B*, then the whole of *B* is copied in parallel into *A*. This enables a fair start, with both players having an identical mess.

From here on the two tables are maintained independently; in one case by use of a finite discipline. In the other case? Who the heck knows what some humans are apt to do. In the study that follows we describe how the program works so that you can get your own copy running. From that point perhaps you can figure out how some human's work.

WHAT'S INSIDE OF INVERT

A glance at the top area of the listing shows the usual program initializing code and the option to display the instruction repertoire. A "funny" is contained within the rules sequence. The player is asked whether he understands the rules; if he answers yes, the game gets under way. If a no response is detected by statement 1160 the message constants in lines 1190 and 1200 are output; then a return to the mainline occurs, just as would have been the case anyway.

At the reentry point (line 90) the *Q$* (for query) field is conditioned with an asterisk character. This is done so that later, while in the end-of-game dialog, if the content of *Q$* is overlaid the mainline can detect that a game has been finished and alter the recycle addressing accordingly. Alternative routing is at the bottom of the mainline, within statements 180 through 210.

A game is started out of line 100 with a jump to the module that is used to set up the strings initially. The short sequence from line 2000 to line 2140 does this, with the RETURN bringing the program back to the primary mainline. What happens within the string setup module deserves a brief description.

A typical FOR-NEXT loop spreads the *I* variable down through *table A*. This coding is in lines 2001, 2010, and 2020. Another loop, from line 2030 to line 2100, moves the numbers from *table A*, in a helter-skelter manner, throughout *table B*. A combination of the INT and the RND function is used to generate a number in the zero to nine range. Since only one to nine is wanted any zero that is fetched up is discarded by the branch in line 2050.

The generated integer is used, then, to pick up one of the "canned" numbers from *table A*. That number is moved to *table B*, sequentially from the top, using the loop control variable as a subscript. To preclude any duplicates being created in the second table, as each number is retrieved from the *table A* stack, a zero is written into the spot from which it is taken. That is the reason for the test statement in line 2070; any attempt to fetch a zero is ignored, forcing another call to the random-number expression in statement 2040.

The final act in setting up the initial strings is accomplished by the short loop from line 2110 to line 2130. This FOR-NEXT sequence merely copies *table B* over to *table A*, resulting in a matched pair of strings. That's all that is required, so the RETURN in line 2140 is next executed, which does take us back to the mainline at line 110. From there we fall into a series of GOSUB instructions, which collectively are a task dispatching sequence.

A complete round of play is accomplished by the series of jump statements that extend from line 120 to line 160. The conditional testing expression that is inserted at line 135 has to do with the elective forfeiture option. If *Q* does contain a zero at this point the regular cycle is abandoned in favor of getting to the end-of-game logic quickly. Since the normal flow does follow the list of jumps as presented, that is the sequence in which those task modules will be described.

Display 2 Tables (3000—3140)

The DATA constants (00, 01, 02, etc.) are primarily for including a parenthetical notation of the round number each time the strings are output. The coding that displays these numbers is in line 3030. There is a secondary use of the first constant (00). Later, in the module that accomplishes the computer's move, if *T$* has the double-naught code in it a new game is signified.

The list of eighteen numbers is supposed to be sufficient, meaning the maximum number of turns the computer should ever need to sequence its string is eighteen. The trailing asterisk character is included to trigger a read-restore function should it ever be needed. Normally it won't be. During debugging, however, if a vectored start is made directly into a procedural area this logic will insure that the READ pointer never gets out of logical range.

The balance of this subroutine is essentially just the two loops that print the strings. One starts in line 3060, and it prints the *table A* string followed by the YOURS label. The other loop, which begins in 3100, works the same way to print *table B*, followed by MINE.

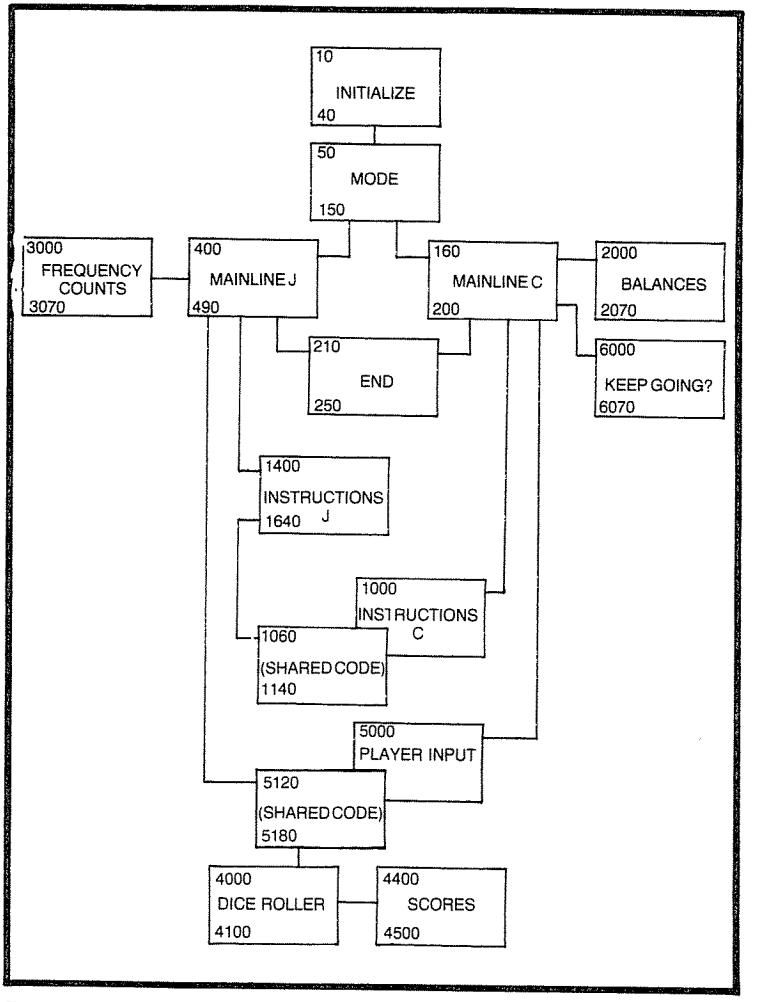


Fig. I-2. Program template for Invert.

Player Input (4000—4340)

Basically there are three services provided by this subroutine: player input is accepted (line 4030), qualified (lines 4040 and 4050), and either the move is approved or a forfeiture condition is set up. The *E* variable that is preconditioned in line 4001 is used by the invalid-entry logic that spans from line 4130 to line 4250. Three different errors are possible by this series in the event repeated inputs fail to pass the validity checks. Just a bit of humor here that most players will never see, save those wanting to test your programming finesse.

Notice that, before zero is declared an error, the sequence from 4300 to the bottom of the module is used to qualify whether the input of zero is intentional. If a forfeiture is being declared when the RETURN in 4120 is branched to the zero will be in *Q* and *Q$* will have a Y code in it.

Invert Player's Numbers (6000—6090)

Here is what the game is all about; yet only ten statements are needed to reverse any portion of the player's string. The process begins with a copy statement to move the first field in *table A* to the collector table. This statement (line 6010) is within a FOR-NEXT loop that is defined for a maximum of nine loops, but it seldom has to run that long. In any event, as each table element is compared with the player's move (saved in *Q*), when a match does occur, as it must at some point, the loop is broken with a branch to line 6040.

When that happens whatever is in the loop variable (*I*) denotes the length of the substring that is to be reversed. Another FOR-NEXT loop is started then, using *J* as a a counter. Transfers from *C* back to *A* begin then, using the *I* variable for the pickups and the *J* counter as the subscript for putting the numbers back into *table A*. During this loop, as *J* is incremented, *I* is decremented by statement 6060, the two working in conjunction to effect the switch in the string's sequence. When *I* has finally been decremented through one, the zero test in line 6070 will learn that the reversal is complete and the routine can be exited.

Computer's Pick (5000—5230)

Conceptually this module is much the same as the previous one. In this case there are two pairs of the FOR-NEXT loops that do the reversal process, and *table B* and *table C* are the ones used (instead of *A* and *C*). Other additional statements have to do with deciding what the machine's move ought to be, based on the two-step cycling required by the driving algorithm. There is also the elementary logic needed to keep track of the next digit to be worked on, depending on a serial descent from nine.

The countdown, as predicted earlier, is conditioned off of the 00 code as can be seen in statement 5001. Variable *X* is used by this module to contain the digit that the machine is going for. The second pair of switcher loops that extend from line 5130 to line 5230 are used for step 2 of the process. The earlier pair (lines 5020 to 5120) are used if the "pick" is not yet in the string's leftmost position. The distinction as to which step is needed is argued by the conditional expression in line 5030. If the first table element has in it the number

that is needed it's the second step that should be done, even though the string may have been initially generated such that step 1 is unneeded.

As a final act of using the step 2 sequence, in every case the *X* value is decremented by line 5220 so that the next-required digit is known when this module is again called upon.

It was mentioned in the overview that these procedures will not take advantage of any natural ordering of a string. There is one slight exception to that statement. Notice that it is possible—though it doesn't happen very often—if the jumbled string is presented in just the right sequence when this module is called, if *X* was just decremented and it agrees at that point with the value in the table's first spot, two successive iterations of step 2 can occur.

Check for a Winner (7000—7280)

A quick glance at this final major module reveals that it is largely just dialog having to do with the end-of-game logic. The first win test that is made is done by the loop that runs from line 7006 to line 7020. A parallel comparison is made of the human's string with the loop's counter. Any nonmatch indicates a break in an assumed serially ascending sequence; so a branch is taken to a comparable loop that starts in line 7210. There the same kind of logic is used to check the machine's string table.

Before either of these tests can get under way the check in line 7002 may find a zero-forfeit code in *Q*, in which case neither test is made. They would probably fail anyway. The only other real service provided by this module is the addition of a one to either *W1* or *W2* if a win has been determined either by logical qualification or by player default.

If neither is the case a return to the mainline is in order to permit another round of play. In any event, if the program is to continue to run, a RETURN does occur back to the mainline. There the determination as to whether the game did end while in this module is made on the basis of whatever is in *Q$*. If asked for and received the *Y* code implanted in *Q$* by line 7100 will appear as a non-match to the asterisk character that was stored there by the game's housekeeping logic.

Looking back through the program listing for *Invert*, it is noticeable that much of it is operator dialog. Most of the mechanics on the other hand are fairly compact. So, while it may be a bit boring to code and load, it shouldn't take long to get it running correctly—all of which ought to be worth the effect.

THE PROGRAM

```
 10 REM  "INVERT"
 20 REM
 30 GOSUB 9000
 40 DIM A(9), B(9), C(9)
 50 PRINT "WANT INVERT RULES (Y OR N)";
 60 INPUT Q$
 70 IF Q$ <> "Y" THEN 90
 80 GOSUB 1000
 90 LET Q$ = "*"
100 GOSUB 2000
110 PRINT
120 GOSUB 3000
130 GOSUB 4000
135 IF Q = 0 THEN 160
140 GOSUB 6000
150 GOSUB 5000
160 GOSUB 7000
180 IF Q$ = "*" THEN 110
190 RESTORE
200 PRINT
210 GOTO 90
1000 REM  "INSTRUCTIONS FOR INVERT"
1001 PRINT "INVERT: YOU AND I COMPETE."
1010 PRINT "WE START WITH JUMBLED NUMBERS"
1020 PRINT "LIKE THIS:"
1030 PRINT " 4 3 2 1 5 6 7 8 9 "
1040 PRINT "YOU SPECIFY HOW MANY NUMBERS"
1050 PRINT "TO INVERT (TRYING TO GET TO"
1060 PRINT "123456789 BEFORE I DO.)"
1070 PRINT "THE STRING TO BE INVERTED STARTS"
1080 PRINT "ON THE LEFT, AND GOES TO THE"
1090 PRINT "NUMBER YOU INPUT - LIKE THIS:"
1100 PRINT " 4 3 2 1 5 6 7 8 9 "
1110 PRINT "INVERT 4 - ?"
1120 PRINT "IF YOU TYPE A 1, THEN YOU GET"
1130 PRINT " 1 2 3 4 5 6 7 8 9 "
1140 PRINT "UNDERSTAND (Y OR N)";
1150 INPUT Q$
1160 IF Q$ = "N" THEN 1190
1170 PRINT
1180 RETURN
1190 PRINT "THEN LET'S TRY A GAME OR TWO,"
```

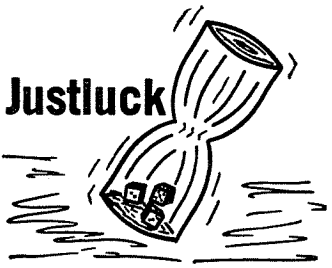
```
1200 PRINT "-- YOU'LL CATCH ON."
1210 GOTO1170
2000 REM  "BUILD MATCHING TABLES"
2001 FOR I = 1 TO 9
2010 LET A(I) = I
2020 NEXT I
2030 FOR I = 1 TO 9
2040 LET J = INT(10*RND(1))
2050 IF J < 1 THEN 2040
2070 IF A(J) = 0 THEN 2040
2080 LET A(J) = 0
2090 LET B(J) = I
2100 NEXT I
2110 FOR I = 1 TO 9
2120 LET A(I) = B(I)
2130 NEXT I
2140 RETURN
3000 REM  "DISPLAY 2 TABLES"
3001 DATA 00, 01, 02, 03, 04, 05, 06, 07
3002 DATA 08, 09, 10, 11, 12, 13, 14, 15
3010 DATA 16, 17, 18, *
3020 READ T$
3030 PRINT "("T$")"
3040 IF T$ <> "*" THEN 3060
3050 RESTORE
3060 FOR I = 1 TO 9
3070 PRINT A(I);
3080 NEXT I
3090 PRINT "YOURS"
3100 FOR I = 1 TO 9
3110 PRINT B(I);
3120 NEXT I
3130 PRINT " MINE"
3140 RETURN
4000 REM  "PLAYER INPUT"
4001 LET E = 0
4010 LET Q = 0
4020 PRINT "INVERT"A(1)"- ";
4030 INPUT Q
4032 LET Q = INT(Q)
4034 IF Q = 0 THEN 4300
4040 IF Q < 1 THEN 4130
4050 IF Q > 9 THEN 4130
```

```
4060 FOR I = 1 TO 9
4070 IF Q <> A(1) THEN 4090
4080 GOTO 4130
4090 IF I = Q THEN 4120
4100 NEXT I
4110 GOTO 4130
4120 RETURN
4130 PRINT "ERROR  ";
4140 LET E = E+1
4150 IF E = 4 THEN 4240
4160 IF E = 3 THEN 4220
4170 IF E = 2 THEN 4200
4180 PRINT
4190 GOTO 4010
4200 PRINT "COME ON NOW"
4210 GOTO 4010
4220 PRINT "DO IT RIGHT!"
4230 GOTO 4010
4240 PRINT "#$%&*# - STOP IT!!"
4250 GOTO 4000
4300 PRINT "ZERO?"
4310 PRINT "WANT TO FORFIET (Y OR N)";
4320 INPUT Q$
4330 IF Q$ <> "Y" THEN 4040
4340 GOTO 4120
5000 REM  "COMPUTER'S PICK"
5001 IF T$ <> "00" THEN 5020
5010 LET X = 9
5020 FOR I = 1 TO 9
5030 IF B(1) = X THEN 5130
5040 LET C(I) = B(I)
5050 IF B(I) = X THEN 5070
5060 NEXT I
5070 FOR J = 1 TO 9
5080 LET B(J) = C(I)
5090 LET I = I-1
5100 IF I = 0 THEN 5120
5110 NEXT J
5120 RETURN
5130 FOR I = 1 TO 9
5140 LET C(I) = B(I)
5145 IF I = X THEN 5160
5150 NEXT I
```

```
5160 LET I = X
5170 FOR J = 1 TO 9
5180 LET B(J) = C(I)
5190 LET I = I-1
5200 IF I = 0 THEN 5220
5210 NEXT J
5220 LET X = X-1
5230 RETURN
6000 REM  "INVERT PLAYER'S NUMBERS"
6001 FOR I = 1 TO 9
6010 LET C(I) = A(I)
6020 IF A(I) = Q THEN 6040
6030 NEXT I
6040 FOR J = 1 TO 9
6050 LET A(J) = C(I)
6060 LET I = I-1
6070 IF I = 0 THEN 6090
6080 NEXT J
6090 RETURN
7000 REM  "CHECK FOR A WINNER"
7002 IF Q = 0 THEN 7270
7006 FOR I = 1 TO 9
7010 IF A(I) <> I THEN 7210
7020 NEXT I
7030 PRINT "CONGRATULATIONS"
7040 LET W1 = W1+1
7050 PRINT "SCORES: YOU ="W1;
7060 PRINT "   ME ="W2
7070 PRINT
7080 PRINT "GO AGAIN (Y OR N)";
7090 INPUT Q$
7100 IF Q$ = "Y" THEN 7190
7110 IF W1 < W2 THEN 7150
7120 IF W1 > W2 THEN 7170
7130 PRINT "THANX FOR THE GAME"
7140 END
7150 PRINT "CHICKEN!!"
7160 GOTO 7130
7170 PRINT "GOOD - YOU'RE TOO CLEVER ANYWAY"
7180 GOTO 7130
7190 RESTORE
7200 RETURN
```

```
7210 FOR I = 1 TO 9
7220 IF B(I) <> I THEN 7200
7230 NEXT I
7240 PRINT "HA HA"
7245 GOSUB 3100
7250 LET W2 = W2+1
7260 GOTO 7050
7270 PRINT "THANK YOU"
7280 GOTO 7250
9000 REM  "RANDOM NUMBER ROUTINE"
9010 LET Z = RND(1)
9020 RETURN
```

J



Here is a dual-purpose program. The first purpose is a game called *Chuck-a-Luck*. It's a fast-moving game played with three dice, and a one-time favorite with riverboat gamblers. The dice, enclosed in an hourglass-shaped wire cage, are rolled when the dealer turns the cage over. Bet on any number, one through six. If your number comes up on one of the dice you win. You win double if your number comes up on two dice, and triple if it comes up on all three.

This game program simulates the part about turning over a cage, of course. We have found it difficult to turn over some models of microcomputers. Otherwise, even the legendary Bat Masterson would recognize this game as a direct takeoff of *Chuck-a-Luck*.

Why *Justluck* then? A second purpose of this program enables you to prove that the random-number generating feature of your computer really does involve just luck. Besides, *Chuck-a-Luck* is too long a name to use as an eight-character program label, and it doesn't start with the letter J.

The dual aspect of this program includes a startup sequence that has three pages of operator instructions. The first offers the choice as to which mode to run in. Either the usual game of Chuck-a-Luck may be selected or the alternative may be the benchmark mode to ascertain the degree of natural odds provided by computerized games. After selecting either the C or the J mode—the symbolic codes are self-explaining—a second page of instructions may be optionally printed; depending on which mode was picked.

In the *Chuck-a-Luck* mode the game runs as described earlier. The player inputs a bet, the dice are rolled, and the winnings are

computed. The house keeps all lost bets in a counter and the player's balance is maintained in another. On every tenth roll these counters are displayed, and on every twentieth roll the game may be ended or continued, optionally.

The *Justluck* mode is basically the same process, except the cage is continuously turned without stopping for a bet each time, and a tally is made of the three types of match conditions. There are separate counters for singles, doubles, and triples. At the start, as is explained in page 3 of the instructions, four questions are asked of the operator. They are:

(1) BET?

(2) NUMBER?

(3) ROLLS?

(4) PRINT?

Questions 1 and 2 are common to the regular game: the first expects a dollar-amount entry, and the second requires a number of from one to six. The third and fourth questions are unique to the J mode. Question 3 (Rolls?) anticipates a whole number to be used for a run-until-done limit. When the limit is reached, the program will break out of the otherwise self-perpetuating loop at the normal end-of-game halt point. At this time the contents of the singles, doubles, and triples counters are output, as well as the house and player's balances.

Question 4 (Print?) is supposed to be answered with *Y* or *N*, meaning yes (print each roll) or no (suppress all printing until the end of the run). The no-print option is intended to benefit those suffering from an austere computing budget. If you'd rather not have umpteen feet of printer paper wasted for, say, a thousands rolls of the dice, the N response to the fourth question is warranted. For short runs or for CRT-based output, the turns can be fully displayed. In this case the values of the dice are printed, followed immediately with the accumulations in the winnings counters and the frequency counters.

JUSTLUCK'S ARCHITECTURE

A glance at the illustration in Fig. J-1 shows how this program achieves its dual personality. Two separate mainlines are used to run in either the J mode or the *C*-mode. Those subroutines that are exclusive to one or the other are shown out to the sides of either, and most everything else is parrallel below, in the middle.

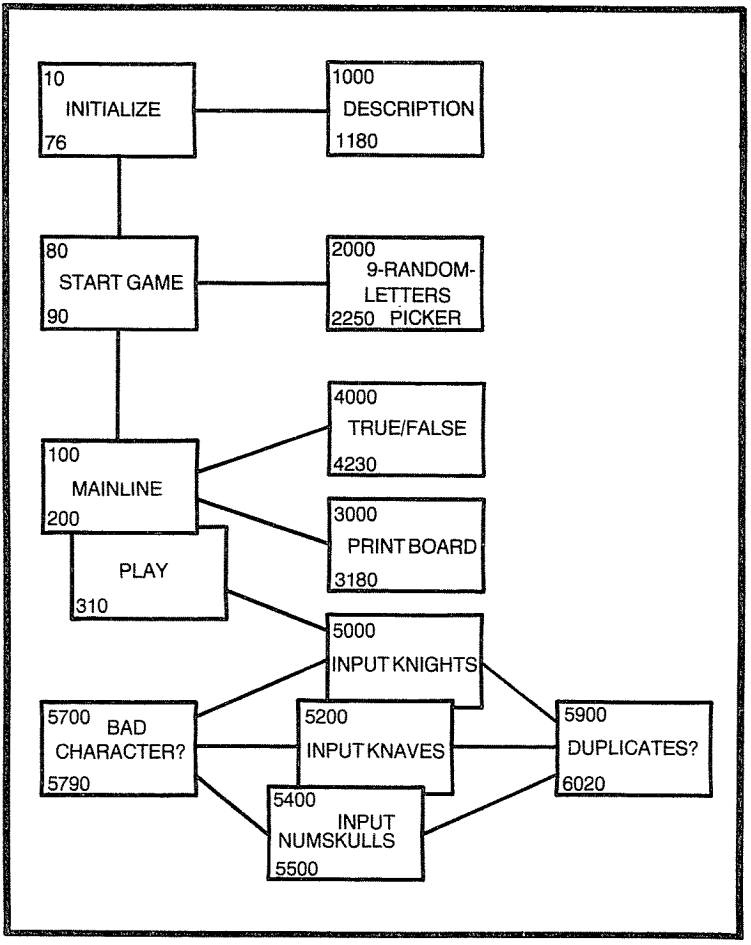


Fig. J-1. Program template for Justluck.

Notice especially how the instructions and the input areas are joined. Often subroutines are accessed from multiple points in a program. In this study this is true, also, with an additional attribute. The instructions C module and the player input modules were constructed in such a way that sharable statements are located near the bottom of the routines. This allows jumping into the routine midway down from the top while the J mode is in force. And out of the shared-code portion of the input subroutine another two levels of jumping serve both modes, to roll the dice and to affix their individual scores.

All RETURN statements effect relinking back to the calling mainline in any event. Those characteristics that are most notice-

able to the operator, depending on the mode then running, are largely established within the respective mainline sequences. Oddly enough, however, neither of the mainlines contains very many statements. This will be seen in the following study of the *Justluck* program listing.

JUSTLUCK'S LISTING

Two dimensioning expressions in line 40 reserve work space for the dice (D), and for the match counts (M). This is all there is to the business of globally initializing the program. From line 50 down through line 150 the mode option is established. Notice that the mode code (C or J) is accepted into *M$* for storage purposes. Later, when it is necessary to know which mode is in force, this variable can be tested.

The appropriate dispatch to the selected mainline is done from line 120 or line 130. Because *Chuck-a-Luck* is the simpler of the two, it is the first offered for study.

Chuck-a-Luck

All of the subroutine calling for the regular dice game is done by the series from line 160 to line 200. The conditional jump to line 1000 out of line 160 is triggered on the basis of the C counter having zero in it. It is the C counter that is used for knowing which roll is being bet on, and in the beginning it does contain zero. The conditional jump takes place then to output the game's brief instructions. The C value is never again zeroed unless the player switches modes. The startup of a J run will clear the counter; so the instructions will always be printed just prior to playing Chuck-a-Luck. The remainder of the nucleus of this mainline is unconditionally executed.

Statement 165 force-loads an *N* into *Q$* so that all end-run tests can presume the status of this variable. At various places in the program the player is asked whether he wishes to continue playing. The questions are always phrased so as to elicit a Y response as an exception to going on. An obvious use of this technique can be seen in the looping conditional in line 200.

Three intervening GOSUB statements in lines 170, 180, and 190 do the jumping required to support running in the C mode. In the order called those tasks accept player input, roll the dice, update the balances, and print them when conditions permit.

Dialog for entering a bet and choosing a number to try to match is in the sequence from line 5000 to line 5180. Mechanics there are easily understood with a definition of the symbols used. The *B* and *N* workers are used for player inputs for bets and numbers. In line

5150 the field called *P* is updated with the player's winnings. This arithmetic expression does a multiplication of the bet by the number in the match counter (*M*). This works to give credit for doubles and triples, and even for singles. A single match is worth only one times the bet. If *M* has a zero in it, meaning there were no dice matching the player's *N* entry, line 5170 is used to give the bet amount to the house (*H*). The dice roller was called earlier, from line 5120, so that routine should be looked at next. It is there, also, that the *M* count is established.

Before the dice are rolled by the routine that ends in line 4100 the first statement in line 4000 initializes *M* with zero. As the dice are each generated they are compared to *N* in line 4040, and if appropriate *M* is incremented by one in line 4050. All of this is inside the three-times loop that calls the six-statement subroutine, lines 4400 through 4500. This subroutine generates a number between one and six and returns to the roller loop.

As the dice are rolled each time through the loop the *M* value is also used to add a one to the frequency counters in *table M*. These counters are only output when the program runs in the *Justluck* mode, but the frequencies are tallied while running in either mode.

Back to mainline C. The short subroutine from line 2000 to line 2070 is called upon next, first to find out whether it is time to print the player's balances, and second to do the printing if the rounds counter is an even multiple of ten. Either way, the RETURN in line 2070 goes back to the mainline for the final jump—this time to line 6000.

The question the program is asking itself here is whether the rounds counter is a multiple of twenty. To learn this easily a repeating subtraction trick is used, continuing until the copy of *C* that is in the temporary worker (*T*) falls exactly on or below zero. If zero is passed the RETURN is immediately executed. If at some point *T* is reduced to exactly zero by the successive subtractions of twenty, a stop question is asked. The response is accepted into *Q$* for action as directed back in the mainline. An N response, as described earlier, will allow the play to continue.

If the player overlays a *Y* into *Q$* the dialog from line 210 to line 230 will allow an optional switch in the running mode. For the sake of this study assume the branch to line 50 is taken. This lets us look at *Justluck*.

Justluck

Because the *J* mode takes advantage of *Chuck-a-Luck's* coding wherever possible, it is only necessary to narrate the differences.

The mainline from line 400 to line 490 controls most of the *Justluck* flow, beginning with an immediate jump to line 1400.

Another GOSUB is executed to line 1060 to use the housekeeping logic that is in the tail end of the instructions C module. The first-level RETURN will complete a startup of *Justluck*, extending downward from line 1410 to line 1640. Dialog here is for conditioning a run limit (*L*) and to load *Q$* with a code for controlling the no-print option.

Each lap through the mainline checks whether the rounds counter (C) has yet caught up to the *L* (for limit) value. It is the GOSUB 5120 in line 420 of the mainline that causes the dice to be rolled, using all of that beyond line 5120 just as is done by *Chuck-a-Luck*. The remaining difference while running in the J mode is the use of the routine at lines 3000 through 3070.

This is the only complete module that is exclusive to *Justluck*. It is merely a linear series of PRINT statements to print all of the program's counters. Access to this supporting subroutine is conditional. In one case if printing is supposed to be suppressed line 440 is bypassed. If *Q$* has a *Y* in it the conditional in line 430 will permit a jump to line 3000 each time the mainline is cycled. On the other hand the jump to line 3000 is done from line 462 when a *J* run is completed.

There is one other nuance exclusive to the J mode. It is well within the area of shared programming, in the area of line 5130. This is the line that actually prints the dice. This PRINT statement is conditionally hit based upon the two preceding statements. A *C* mode indicator in *M$* forces the printing, but if *M$* has a *J* in it the next test is done instead. Line 5124 acts on whether *Q$* has *N* in it, optionally put there by question 4 ("PRINT?") that was asked when *Justluck* was initialized. This is how you can save paper on those long runs: nothing will be printed until the run ends.

It is this feature, perhaps, that best justifies your adoption of this program into your library. When your machine has nothing better to do you can let it play all by itself. I once started mine at lunch time. When interrupted at sundown the rounds counter was only a little over 2000. Imagine the fun the computer would have had if I had let it continue until the 5000-time limit that was specified.

THE PROGRAM

```
10 REM  "JUSTLUCK"
20 REM
30 GOSUB 9000
40 DIM D(3), M(3)
50 PRINT "THERE ARE 2 MODES:"
```

```
 60 PRINT "   (1)  CHUCK-A-LUCK"
 70 PRINT "   (2)  JUSTLUCK
 80 PRINT
 90 PRINT "INDICATE YOUR CHOICE"
100 PRINT "(C OR J)";
110 INPUT M$
120 IF M$ = "J" THEN 400
130 IF M$ = "C" THEN 160
140 PRINT "YOU MUST SELECT A MODE"
150 GOTO 80
160 IF C = 0 THEN GOSUB 1000
165 LET Q$ = "N"
170 GOSUB 5000
180 GOSUB 2000
190 GOSUB 6000
200 IF Q$ = "N" THEN 170
210 PRINT "WANT TO STOP (Y OR N)";
220 INPUT Q$
230 IF Q$ = "N" THEN 50
240 PRINT "END OF PROGRAM"
250 END
400 GOSUB 1400
410 IF C = L THEN 460
420 GOSUB 5120
430 IF Q$ = "N" THEN 450
440 GOSUB 3000
450 GOTO 410
460 IF Q$ <> "N" THEN 465
462 GOSUB 3000
465 PRINT "ANOTHER RUN (Y OR N)";
470 INPUT Q$
480 IF Q$ <> "N" THEN 400
490 GOTO 210
999 REM  "INSTRUCTIONS-C"
1000 PRINT ". . . CHUCK-A-LUCK . . ."
1010 PRINT "YOU BET AN AMOUNT AND PICK"
1020 PRINT "A NUMBER (1-6).  IF YOUR NUMBER"
1030 PRINT "COMES UP YOU WIN.  PAIRS PAY"
1040 PRINT "DOUBLE -- TRIPS PAYS 3 TIMES"
1050 PRINT "YOUR BET."
1060 LET P = 0
1070 LET H = 0
1080 LET C = 0
```

```
1090 FOR I = 1 TO 3
1100 LET D(I) = 0
1110 LET M(I) = 0
1120 NEXT I
1130 LET M = 0
1140 PRINT
1150 RETURN
1399 REM  "INSTRUCTIONS-J"
1400 GOSUB 1060
1410 PRINT ". . . JUSTLUCK . . ."
1420 PRINT "BET";
1430 INPUT B
1440 PRINT "NUMBER";
1450 INPUT N
1460 LET N = ABS(INT(N))
1470 IF N < 1 THEN 1490
1480 IF N < 7 THEN 1510
1490 PRINT "ILLEGAL"
1500 GOTO 1440
1510 PRINT "LIMIT";
1520 INPUT L
1530 LET L = ABS(INT(L))
1540 IF L < 2000 THEN 1580
1550 PRINT "ARE YOU SURE (Y OR N)";
1560 INPUT Q$
1570 IF A$ = "N" THEN 1510
1580 PRINT "PRINT (Y OR N)";
1590 INPUT Q$
1600 IF Q$ = "N" THEN 1640
1610 IF Q$ = "Y" THEN 1640
1620 PRINT "I DON'T UNDERSTAND..."
1630 GOTO 1580
1640 RETURN
1999 REM  "BALANCES"
2000 IF C = 0 THEN 2070
2010 LET T = INT(C/10)
2020 IF C <> T*10 THEN 2070
2030 PRINT "ROLL#" C;
2040 PRINT "  YOU=$" P;
2050 PRINT "  ME=$" H
2060 PRINT
2070 RETURN
2999 REM  "FREQUENCY COUNTS"
```

```
3000 PRINT "ROLL#" C;
3010 PRINT "  YOU=$" P;
3020 PRINT "  ME=$" H
3030 PRINT " SGL=" M(1);
3040 PRINT "  DBL=" M(2);
3050 PRINT "   TRP=" M(3)
3060 PRINT
3070 RETURN
3999 REM  "DICE ROLLER"
4000 LET M = 0
4010 FOR I = 1 TO 3
4020 GOSUB 4400
4030 LET D(I) = X
4040 IF N <> X THEN 4060
4050 LET M = M+1
4060 NEXT I
4070 IF M = 0 THEN 4090
4080 LET M(M) = M(M)+1
4090 LET C = C+1
4100 RETURN
4399 REM  "SCORES"
4400 LET X = INT(10*RND(1))
4410 IF X < 1 THEN 4400
4420 IF X < 7 THEN 4500
4430 LET X = INT(X/2)
4500 RETURN
4999 REM  "PLAYER INPUT"
5000 PRINT "BET";
5010 INPUT B
5020 IF B > 0 THEN 5050
5030 PRINT "WHAT? -- ILLEGAL"
5040 GOTO 5000
5050 PRINT "NUMBER";
5060 INPUT N
5070 LET N = ABS(INT(N))
5080 IF N > 0 THEN 5110
5090 PRINT "NIX -- ILLEGAL"
5100 GOTO 5050
5110 IF N > 6 THEN 5090
5120 GOSUB 4000
5122 IF M$ = "C" THEN 5130
5124 IF Q$ = "N" THEN 5140
5130 PRINT D(1)" "D(2)" "D(3)
```

```
5140 IF M = 0 THEN 5170
5150 LET P = P+B*M
5160 GOTO 5180
5170 LET H = H+B
5180 RETURN
5999 REM  "KEEP GOING?"
6000 LET T = C-20
6010 IF T < 0 THEN 6070
6020 IF T = 0 THEN 6050
6030 LET T = T-20
6040 GOTO 6010
6050 PRINT "STOP CHUCK-A-LUCK (Y OR N)";
6060 INPUT Q$
6070 RETURN
9000 REM  "RANDOM NUMBER ROUTINE"
9010 LET Z = RND(1)
9020 RETURN
```

K

Knights

History provides an interesting introduction to the principals in this game. First, there are the knights. In the century or so just past these are supposed to have been the noblest of men—always honest and gentlemanly, but not necessarily horsemen. Flipping back through the pages of history we find that their title has more to do with horsemanship than with honorable virtues. This game presumes the modern connotation, at least to the extent that knights are considered to be incapable of telling a falsehood.

Knaves, on the other hand, are considered today to be inveterate storytellers. In days of old they were merely small boys, boy servants, or young men of humble origin. Isn't it amusing that with the passing of time a guy on a horse has become virtuous and boyhood has become almost synonymous with lying?

Remaining principals in the game of *Knights* seem to have suffered no change in all of history. Numskulls used to be, still are, and probably always will be considered to be unreliable. It is a good idea, in history, in life, and especially in this game, to pay no heed to their advice.

These are the characters in *Knights*. In the beginning there are nine letters displayed, in groups of three, and below each is a letter, *T* or *F*. Two truth symbols are printed as if in response to the question: "Are these now right?" The game's player (that's you) sits in judgment of these declarations. The object of the game is to move the letters about until they are all properly grouped under the headings of knights, knaves, and numskulls (three per).

As the game opens you are forewarned that: (1) knights never lie, (2) knaves always lie, and (3) you can't trust numskulls. Your ability to deduce the correct placement of the individual letters depends on your faith in these axioms.

Any letter appearing in the knights' group that is marked with a *T* is in the correct group. By the same logic an *F* beneath a letter in the first group indicates that the letter is falsely placed. Imperative logic applies to the knaves as well, but their truth indicators must be considered as inverted or switched. Remember: the knaves always lie—you can depend on it.

You cannot depend on the numskulls, however—neither for a correct answer nor even for the same answer twice in a row. The letter *G*, for example, may first appear in the knights' area. At that point assume *G* is shown to be false. If, though, when you retype the letters and move *G* under the knaves, it could now be labeled with a *T* (which must be read as false). If this be the case: *G* must be a numskull. Yet, as you continue to regroup all of the letters, but each time keeping *G* with the numskulls, his truth indicator may alternate between *T* and *F*, sporadically. You just can't trust a numskull.

A reverse trail of markers must be seen to fully appreciate how it is that you can succeed, possibly even in fewer turns than your friends. Regardless of the marker appearing under a letter in the numskulls' area, if, when that letter is moved to the first area it is then tagged as true, it is a knight. Similarly, if a letter is marked with an *F* whenever it is located under the knaves heading, keep it there; it is where it belongs.

The logic of *Knights* is such that when the knights and the knaves are properly grouped, the numskulls will all read as true. The end of the game is reached when you achieve all true answers. Because the knaves are forever liars the indicator string will read; *T-T-T*, *F-F-F*, and *T-T-T*.

When the game does end there is an immediate replay option just like other games in this book. Because of this it should be apparent that the solution to one game has no bearing on a following game. Not only do the letters take on a different labeling, the letters themselves are apt to change from game to game.

Although a game uses only nine letters of the alphabet (all different), the program houses twenty-four letters. The reason that there are not twenty-six possibles is that, the letters *T* and *F* are excluded from the random selection list. This is done to preclude confusion as to what is supposed to be typed for each turn; turns consist of three entry points, with three letters asked for at each.

A round begins with a turn number notation and the knights prompt. Three letters should be typed, then, separated by commas.

When the entry of the knights character string is terminated, the knaves prompt appears. Again, three letters should be entered, terminated, and the numskulls label will appear to permit entry of the remaining group of three letters.

Following the three entries a new board is printed showing your rearrangement of the letters with their resulting truth flags. And so goes the game until the truth of the groupings is correctly established.

The game of *Knights* can be fun—even exasperating for some. The program can be fun too. It can be fun to code and load, and it can be exasperating to get to work correctly as well, especially if it is not lifted accurately from what follows. In the section that deals with the internal logic my original bugs are exposed, as well as how they were exterminated. The design layout survived intact, however, so we will start there.

THE LAYOUT OF KNIGHTS

As to both problem definition and task allocation this program is rather simple. The drawing in Fig. K-1 reflects the organization of the different sections of the program into logical tasks and how they are interconnected.

At the top there is nothing very original. Both the program's initialization and the optional description routines are typical. The game's startup requirements aren't much, either: mostly just a jump to a stand-alone module that selects the nine letters to be used. From there the logical flow is directly into the mainline. As an architecture up to this point this layout is not too exciting.

The presentation sequence of the modules, as connected to the mainline, does have one peculiar attribute. According to the template (and to the actual processing sequence) the truth table is generated from the outset as soon as the newly generated random-letter string is formed.

Because the letters are picked out, their truth is qualified; and then the first board is printed it is possible that an automatic win can occur. Odds on this happening have got to be on the order of being ridiculous, however. My attitude was if this ever does happen so what? You can immediately invoke the replay option to start another game.

Actually I did not have much difficulty with the program's layout. This one was chosen to favor being able to stack the blocks as shown. Notice how all of the play business is nicely separated from the usual mechanics for managing a game program. This does have an advantage during fault isolation exercises, but a careful

study of the program's internals may preclude your having to depend on this advantage.

KNIGHTS FROM WITHIN

The program listing is usual enough down to line 70, including the jump to the optional description routine. Statements 72, 74, and 76 define tables used by the program, so their use ought to be known before we encounter them further.

Table X$ is filled once with the letters of the alphabet. Only twenty-four slots are allocated because the *A* to *Z* DATA string does not include the letters *T* and *F*. *Tables C$* (C for computer) and *P$* (P for player) are a matching pair having nine fields each. The first is for holding the computer's nine randomly selected letters, and the second is for player entries. *Table T*, of course, is for the truth indicators—one for each letter in the player's string.

Definition of the tables is the last of the program initialization steps, and the next two lines (80 and 90) constitute the start game 9 operation. Line 90 merely sets the rounds counter to zero, but the GOSUB 2000 in line 80 deserves some explanation.

The routine that extends from line 2000 to line 2250 is the one that generates the initially scrambled list of nine letters. The process is accomplished through the use of three FOR-NEXT loops that are of the run-until-done type, one following another.

The first loop is the three statements at lines 2030, 2040, and 2050. *Table X$* is loaded with the twenty-four DATA constants in sequence from the top downward, the DATA pointer is restored by line 2060, and the second loop is entered.

From line 2070 to 2170 the nine letters for this game are selected from the master list and loaded into *table C$*. Since nine are needed the loop is set up to run nine times. The RND expression in line 2080 will fetch a two-digit value in the 00 to 99 range. Since zero cannot be used as a valid table subscript line 2090 will force another try until at least a one is obtained.

The conditional in line 2100 tests to see whether the number meets the maximum of twenty-four; if so a branch is taken to line 2130. Any RND call that gives a number in the 25 to 99 range is divided by two otherwise, and the range tests are repeated until an acceptable value is achieved. The *R* value may now be used to pick a letter from the master list.

Statement 2130 does this, and places the selection into the next available spot in *table C$*. Before the loop is allowed to repeat, however, the conditional in line 2150 checks whether a space code was retrieved. (It would have been placed there in a previous loop

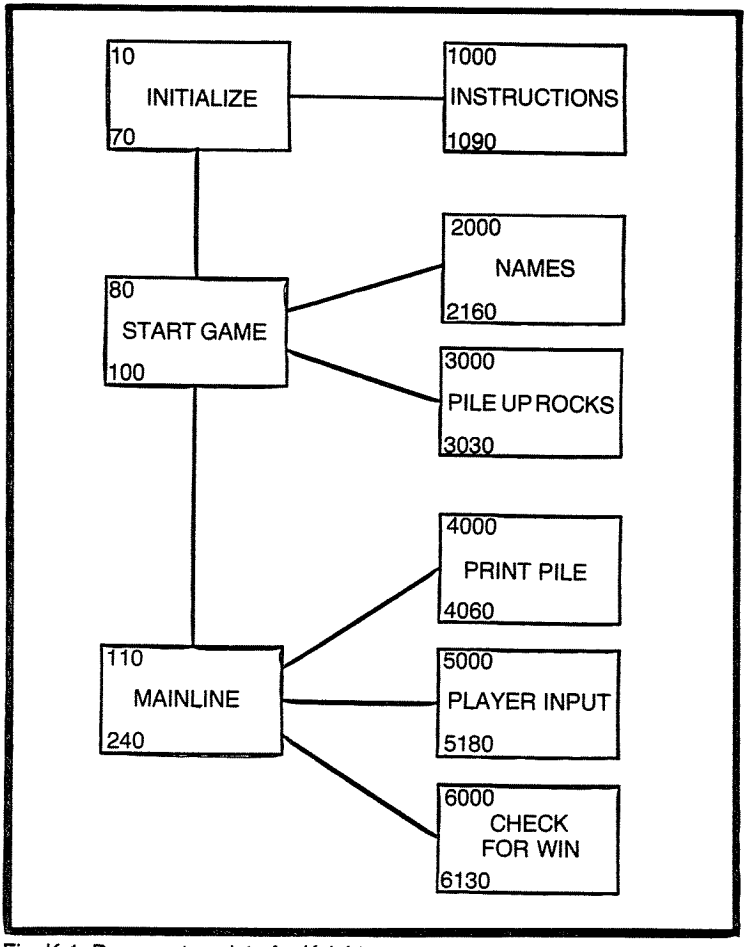


Fig. K-1. Program template for Knights.

by line 2140.) This logic works to preclude any duplicate letters in the final string. Once the RND value is good and a unique letter is placed into the *I*-designated location in *table C$* it is copied into *table P$* by line 2160 and the loop is repeated. Upon completion of nine loops both tables are filled with matching strings of scrambled letters.

Notice that *table P$* contains a winner at this point. Because the program does generate its own solution a temporary patch can be used to find any bugs that manage to crawl into later areas. To take advantage of this self-help opportunity insert a RETURN statement numbered between lines 2170 and 2180. Running of the program with this early exit in place will at least qualify whether or not the

program knows when a game is supposed to be over. Do remember to remove this patch or you'll never get to do anything but win—instantly.

Getting through to the final loop of the scrambler routine is essential to a challenging game. This loop runs nine times, too, to shuffle the player's string. Line 2190 generates a single digit in *R* which is used in conjunction with the *I* counter to swap around the contents of *table P$*. By using the loop control as one of the subscripts every letter is subject to being relocated, although it is possible that some may end up where they started. Either way the result is deemed usable, and the RETURN in line 2250 exits back to the mainline to begin the game.

And a short mainline this one is. The round counter is bumped up by one in line 100 and output in line 110. Two jumps in lines 120 and 130 will get the truth of the present situation and print it all out. The *W* that is tested by line 140 controls the ending of the game (or continuation of play by branching to the GOSUB 5000 in line 300). As long as the win switch remains off player entries are processed by the input modules. The usual looping through the mainline includes jumps to lines 4000, 3000, and then 5000; so that is the order in which they are described below.

From line 4000 to 4230 the truth table is generated afresh, just before each round of play. The object of this routine is to turn on (or off) the *T* or *F* letter indicators and to condition the *W* variable based on the total truth of *table T*. Whether or not a win is in order cannot be learned until the tasks here are finished; so *W* is conditioned to off at the beginning of the module in line 4005. Later, if appropriate, it is set to on with a one.

Within the truth table itself three values are used. For the knights group a *T* is represented by positive one and *F* is zero. Knaves use zero for false also, but a true condition is indicated with a negative one. Rather than looking at the letters in the third group the whimsy of numskulls is simulated by doing arithmetic on the indicators for the letters in the first two groups. This works on the principle that addition of positive and negative ones will sometimes leave a zero, or something, depending on the number of offsetting ones. Because an equal number of positive and negative ones will generate a zero, and an odd number will produce an absolute residue, the numskulls will appear to change capriciously. If a player runs this game long enough he or she might discover the relationships. But few will.

Status of the knights and the knaves is established by two sets of concentric loops that compare the *C$* and *P$* tables. The first pair

of FOR-NEXT loops extends from line 4010 to line 4070. The outside loop picks up the letters from the player's table, one at a time, and the inside loop compares each of them to the first three letters in *table C$*. Just before the comparison is made in line 4040 the *T* field that is pointed to by *I* is made a one. If the comparison fails the loop is simply repeated. If on the other hand an equal match is found the inside loop is broken by branching to line 4060, which resets the one to zero. After three complete laps through the outside loop the program falls through into the next pair of nested FOR-NEXT loops.

The series form line 4080 down to line 4140 is much the same as the previous structure. The fourth, fifth, and sixth letters in the tables are compared here to generate the status codes for the knaves group. Notice in line 4100 that the default setting is zero this time; and if the comparison comes up as equal the *T* field is loaded with negative one. Because of the reversed use of the zero code the knaves will be presented as opposites of knights, meaning they are lying.

Numskulls are codified next, by lines 4150, 4160, and 4170. This loop adds the truth values for the knights and the knaves in parallel, and each sum is placed in the seventh, eighth, and ninth spots in the table. All three groups of letters have now been qualified as to their individual placement, and all that remains is to see if they are all where they belong. Again a loop is used to do this task.

The loop from line 4180 to line 4200 adds together each knights indicator with a corresponding numskulls code and compares the sum with a positive two. Any pair of indicators from the knights and the numskulls that does not equal two is sufficient cause to abandon the test: some character somewhere is not yet properly grouped. When this loop finally does run to completion the game is over; so *W* is set to one by line 4210. Either way the RETURN in line 4230 links control of the game back to the mainline.

From statement 130 in the mainline a jump is made immediately to the routine that prints the board. Three printed lines are output by this series of statements, which extends from line 3000 to line 3180. The first line printed is done by the single statement in line 3020. These are the column headings: KNIGHTS, KNAVES, AND NUMSKULLS. Next a loop is used to print the contents of the player's table in groups of three.

The FOR statement is constructed in line 3080 to cause the loop to run three times. Notice that the *I* variable is incremented by three each time. In doing it this way the print statement in line 3040 can use modified *I* subscripts to output three letters per loop. As

each cycle completes the PRINT command in line 3150 spaces the print buffer to the right for six spaces to cause the next group to be aligned under the next columnar heading. The final PRINT in line 3070 advances the output in preparation for the printing of the truth indicators.

From line 3080 to line 3160 is another FOR-NEXT loop, also incremented by three, to print the *T* and *F* markers. Printing and spacing control is much the same for this loop as the one above, thereby insuring that the markers will align under their corresponding letters. The series of conditional expressions from line 3090 to line 3140, which is inside of the loop, will always print exactly three symbols per iteration of the loop. Whether a *T* or an *F* is printed is determined by the codes in *table T*: a zero always produces an *F*, and either a positive or a negative one will trigger a *T*.

After the print loop runs three times the dummy PRINT in line 3170 completes the output by spacing up once, and the RETURN in line 3180 ends this subroutine.

The last major chunk of this program is all of that having to do with accepting player input. This is all done between lines 5000 and 6020, including learning whether the letters that are typed are usable or not. This does appear to be a long list of instructions; yet there is a considerable amount of redundancy within. Each line of input expects three letters to be typed, one line for the knights, one for the knaves, and a final line for the numskulls. As the lines are entered two jumps will follow to find out whether any of the characters are invalid or whether any on this line or the previous line are duplicates. Study of how this is all done can concentrate on the knights sequence. Logic for taking in the operator's selections for the other two groups works the same.

The INPUT statement in line 5020 expects three letters to be typed, and they must be separated by commas as a convention of BASIC. Two variables are then set up preparatory to use of the error-checking subroutines. The *F* field is used as a flag to know the results of the tests. The flag is always set to one prior to jumping to either test, and if it remains upon return from the test no error was found. Either test module will place a zero into *F* to indicate that the operator must reenter the line just typed.

The other control variable is *L*. This one is used as a limiting base address by the error checkers. It is preloaded with three, six, or nine to indicate the depth of the table scanning that should be done after each line of player entry. The way this is used can be seen by looking at the two test modules.

From line 5700 to line 5790 a pair of nested FOR-NEXT loops are used to check whether each of the player's letters have a

counterpart somewhere in the master string. The subtraction expression in line 5710 adjusts the FROM counter for the *I* loop to run three times, ending with the preloaded limit. The *J* loop is established for a maximum of nine iterations; but as each of the three characters from an input line are looked for in *table C$*, whenever a match *does* occur, the long loop is vacated. If it ever does happen that the full nine comparisons finds no match the character is invalid and *F* is set to zero to flag this condition.

The routine that checks for duplicate characters is similar in concept to the previous one. Two more nested loops are used (lines 5910 through 5970), again using the floating *L* variable. In this case it is *table P$* itself that is examined; instead of always running for a possible nine times it is only permitted to scan down as far as new entries have been made. The reason for restricting the depth of the scan is that characters further down in the table are residue from a previous turn. Notice also that one match is bound to occur in this loop—as the scan passes the letter that is being compared for duplicates. That is why *F* is incremented in line 5950. Because three letters are checked and *F* was initialized with a one, a valid test should end with a four in *F*. The conditional in line 5980 checks for this, and any sum other than a four triggers the error message and resets *F* zero.

That is really all there is to *Knights*. The use of the floating variables within the nested loops can cause some irritation if a minor mistake is made in implementing this coding. Even mine didn't work the first time (nor, alas, the second and third). To get the kinks out I finally resorted to using a temporary loop between lines 3010 and 3020 to dump the *C$* and *T* values prior to starting the input. Knowing what the program was looking for did help.

THE PROGRAM

```
10 REM "KNIGHTS"
20 REM
30 GOSUB 9000
40 PRINT "WANT A DESCRIPTION (Y OR N)";
50 INPUT Q$
60 IF Q$ <> "Y" THEN 72
70 GOSUB 1000
72 DIM X$(24)
74 DIM C$(9), P$(9)
76 DIM T(9)
80 GOSUB 2000
90 LET C = 0
```

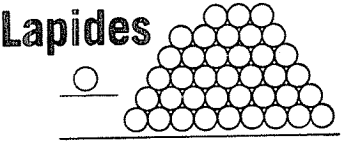
```
100 LET C = C+1
110 PRINT "ROUND #"C
120 GOSUB 4000
130 GOSUB 3000
140 IF W = 0 THEN 300
150 PRINT "GOOD SHOW OLD CHAP"
160 PRINT "WANT TO GO AGAIN (Y OR N)";
170 INPUT Q$
180 IF Q$ = "Y" THEN 80
190 PRINT "SO LONG THEN ..."
200 END
300 GOSUB 5000
310 GOTO 100
1000 REM  "DESCRIPTION"
1010 PRINT
1020 PRINT "KNIGHTS   KNAVES   NUMSKULLS"
1030 PRINT " G,Q,S     R,X,P     L,E,A"
1040 PRINT " T F F     F T F     T T F"
1050 PRINT "THE LETTERS T AND F (TRUE/FALSE)"
1060 PRINT "TELL WHETHER THE LETTERS ABOVE"
1070 PRINT "ARE IN THE RIGHT GROUP.  THE"
1080 PRINT "OBJECT IS TO RETYPE THE LETTERS"
1090 PRINT "TO GET ALL 'TRUE' ANSWERS IN AS"
1100 PRINT "FEW TURNS AS YOU CAN. KNOW THAT:"
1110 PRINT "  KNIGHTS NEVER LIE BUT,"
1120 PRINT "  KNAVES ALWAYS LIE AND,"
1130 PRINT "  YOU CAN'T TRUST NUMSKULLS."
1140 PRINT "LETTERS ARE ASKED FOR IN GROUPS"
1150 PRINT "OF 3 WITH COMMAS (EX:  G,Q,S)."
1160 PRINT "READY"
1170 INPUT Q$
1180 RETURN
2000 REM  "9-RANDOM LETTERS PICKER"
2010 DATA A,B,C,D,E,G,H,I,J,K,L,M
2020 DATA N,O,P,Q,R,S,U,V,W,X,Y,Z
2030 FOR I = 1 TO 24
2040 READ X$(I)
2050 NEXT I
2060 RESTORE
2070 FOR I = 1 TO 9
2080 LET R = INT(100*RND(1))
2090 IF R < 1 THEN 2080
2100 IF R < 25 THEN 2130
```

```
2110 LET R = INT(R/2)
2120 GOTO 2090
2130 LET C$(I) = X$(R)
2140 LET X$(R) = " "
2150 IF C$(I) = " " THEN 2080
2160 LET P$(I) = C$(I)
2170 NEXT I
2180 FOR I = 1 TO 9
2190 LET J = INT(10*RND(1))
2200 IF J < 1 THEN 2190
2210 LET X$ = P$(I)
2220 LET P$(I) = P$(J)
2230 LET P$(J) = X$
2240 NEXT I
2250 RETURN
3000 REM  "PRINT THE PLAYING BOARD"
3010 PRINT
3020 PRINT "KNIGHTS    KNAVES    NUMSKULLS"
3030 FOR I = 1 TO 9 STEP 3
3040 PRINT " "P$(I)","P$(I+1)","P$(I+2);
3050 PRINT "      ";
3060 NEXT I
3070 PRINT;
3080 FOR I = 1 TO 9 STEP 3
3090 IF T(I) = 0 THEN PRINT "F ";
3100 IF T(I) <> 0 THEN PRINT "T ";
3110 IF T(I+1) = 0 THEN PRINT "F ";
3120 IF T(I+1) <> 0 THEN PRINT "T ";
3130 IF T(I+2) = 0 THEN PRINT "F ";
3140 IF T(I+2) <> 0 THEN PRINT "T ";
3150 PRINT "      ";
3160 NEXT I
3170 PRINT
3180 RETURN
4000 REM  "TRUE/FALSE"
4005 LET W = 0
4010 FOR I = 1 TO 3
4020 FOR J = 1 TO 3
4030 LET T(I) = 1
4040 IF C$(J) = P$(I) THEN 4070
4050 NEXT J
4060 LET T(I) = 0
4070 NEXT I
```

```
4080 FOR I = 4 TO 6
4090 FOR J = 4 TO 6
4100 LET T(I) = 0
4110 IF C$(J) = P$(I) THEN 4140
4120 NEXT J
4130 LET T(I) = -1
4140 NEXT I
4150 FOR I = 1 TO 3
4160 LET T(I+6) = T(I) + T(I+3)
4170 NEXT I
4180 FOR I = 1 TO 3
4190 IF T(I) + T(I+6) <> 2 THEN 4230
4200 NEXT I
4210 LET W = 1
4230 RETURN
5000 REM  "INPUT KNIGHTS"
5010 PRINT "KNIGHTS"
5020 INPUT P$(1), P$(2), P$(3)
5030 LET F = 1
5040 LET L = 3
5050 GOSUB 5700
5060 IF F = 0 THEN 5010
5070 GOSUB 5900
5080 IF F = 0 THEN 5010
5200 REM  "INPUT KNAVES"
5210 PRINT "KNAVES"
5220 INPUT P$(4), P$(5), P$(6)
5230 LET F = 1
5240 LET L = 6
5250 GOSUB 5700
5260 IF F = 0 THEN 5210
5270 GOSUB 5900
5280 IF F = 0 THEN 5210
5400 REM  "INPUT NUMSKULLS"
5410 PRINT "NUMSKULLS"
5420 INPUT P$(7), P$(8), P$(9)
5430 LET F = 1
5440 LET L = 9
5450 GOSUB 5700
5460 IF F = 0 THEN 5410
5470 GOSUB 5900
5480 IF F = 0 THEN 5410
5490 PRINT
```

```
5500 RETURN
5700 REM  "BAD CHARACTER CHECK"
5710 FOR I = L-2 TO L
5720 FOR J = 1 TO 9
5730 IF P$(I) = C$(J) THEN 5780
5740 NEXT J
5750 PRINT "ILLEGAL CHARACTER"
5760 LET F = 0
5770 RETURN
5780 NEXT I
5790 RETURN
5900 REM  "CHECK FOR DUPLICATES"
5910 FOR I = L-2 TO L
5930 FOR J = 1 TO L
5940 IF P$(I) <> P$(J) THEN 5960
5950 LET F = F+1
5960 NEXT J
5970 NEXT I
5980 IF F <> 4 THEN 6000
5990 RETURN
6000 PRINT "DUPLICATES -- NO FAIR"
6010 LET F = 0
6020 RETURN
9000 REM  "RANDOM NUMBER ROUTINE"
9010 LET Z = RND(1)
9020 RETURN
```

L



Time again for a touch of class. The name of the game is Latin for *rocks*. In the game there are forty-five rocks, and they are piled up so that the bottom row has nine, the next row up has eight, the next seven, and so on.

Two players compete in playing *Lapides*. They take turns removing rocks from the pile, working from the top down. In a turn, a player may remove as many rocks from the upper row as he or she likes, with one stipulation: he or she must not specify the removal of more rocks that the total remaining in the topmost row. The object is to force the other person to grab the last rock.

This is a fast game and one that is especially enjoyed by children. The listing here has only 107 lines of programming, and this can easily be reduced to less than a hundred if you omit the REM statements. Even if you are clumsy on the keyboard hardly an hour should be required to load and debug this simple program.

Study of how the program is built and how it works should not take long, either. The program template in Fig. L-1 shows all of the parts of the program and how they are conceptually connected. A classic, actually.

LAPIDES LOGIC

The DIM statement in line 25 sets aside a work table to hold the rocks. The rest of the program's cold-start initialization extends down to line 70. Within this area are the usual mechanics for permitting an optional display of the game's brief rules.

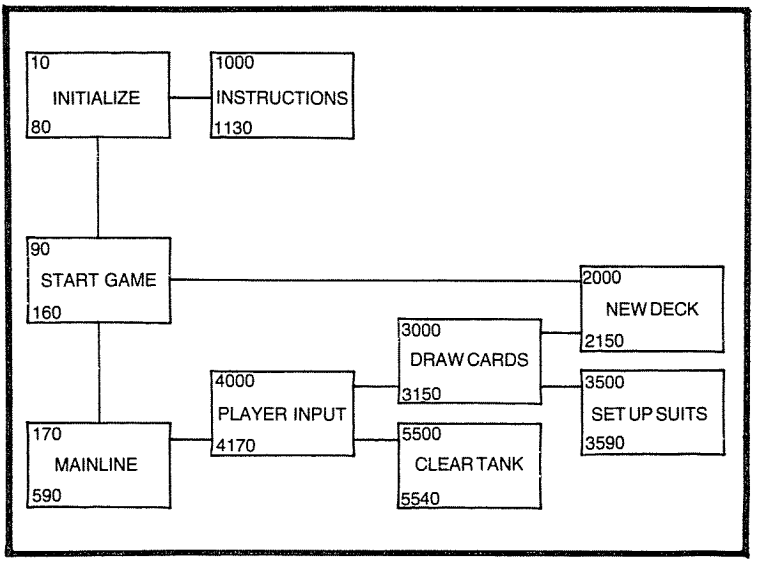


Fig. L-1. Program template for Lapides.

The starting point for repeated games of *Lapides* is line 80. The two jumps, one to line 2000 and the next to line 3000, are for game-level housekeeping chores. The subroutine from line 2000 to line 2160 permits the players to introduce themselves to the program. Whatever two names are typed are stored in *P1$* and *P2$*. During play the players are summoned to the keyboard by their names only; so the logic here insures that two visibly different values are entered.

Statements from line 2110 to line 2160 also provide a bit of programming whimsy. In line 2110 the *P* worker (player) is loaded with a random integer, and in line 2120 it is tested for content. In essence this is like flipping a coin. If the random digit is one of the five in the zero to four range *P* is loaded with a one. There are five digits in the five to nine range also; so the odds are equal that *P* may be initially started with a two. It is the *P* number that is used in the input task to condition which player's name is printed each turn.

The other start-game task is done by the tight FOR-NEXT loop in lines 3000 to 3030. This simply loads the rock table with the numbers 1 through 9, one number per field. As the game proceeds the number indexed by the player is subtracted from these fields, beginning with the top one (a one). As each field is reduced to zero, then, that field is effectively deactivated (until the next game).

As can be seen in the conditional test in line 140 repeated laps through the mainline include the jumps to lines 4000, 5000, and

6000. These are to the supporting routines for printing the decreasing rock pile, the player entries, and the test to see whether the game is over. As long as the *X* variable is returned with more than a one in it the game continues.

The balance of the mainline area contains the go-again dialog, and an option to introduce a new pair of players. Recall that it is in the introduction module that the decision as to who is first is made. If the same two players do continue to play their turn-trading continues right on through from one game to the next.

Print Lapides

A pair of nested FOR-NEXT loops are contained in the routine from line 4000 to line 4060. The outside loop (*I*) controls the number of rows to be printed; the inside loop (*J*) controls the printing of rocks per row. Each time the *J* loop activates it is conditioned to run until whatever limit is in *table R* at the field then referenced by the *I* counter. Notice also the patch statement added as line 4005. This conditional causes the bypassing of the empty rows so that no line spacing will occur until the active portion of the table is encountered.

Player Input

Depending on the *P* number (one or two), the PRINT statement in line 5020 or line 5040 serves as the player prompt, by name. The input in *Q* is then checked in line 5070 to insure that it is some positive integer. It would not be fair to permit a player to pass with a zero entry.

The loop in lines 5110, 5120, and 5130 runs down through *table R* quickly to find the currently active row. The *Q* value is then tested for its reasonableness. (An entry like 99999 would be ridiculous, and the rules of the game do not permit a number larger than the remaining rocks per row.)

Check for Win

The test itself is pure simplicity. The loop that is expressed in lines 6010, 6020, and 6030 does a down-total of the table. The sum in *X* is then compared to a one. A hasty RETURN is taken out of line 6080 as long as the table has more than one rock in it. When finally only one rock remains, the player whose turn it is, is the loser. Hopefully, *it wasn't you*.

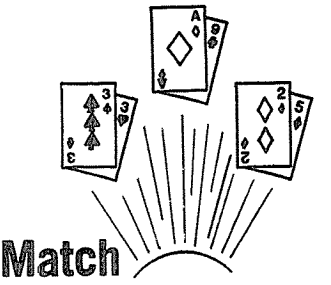
THE PROGRAM

```
 10 REM  "LAPIDES"
 20 REM
 25 DIM R(10)
 30 PRINT "KNOW THIS GAME (Y OR N)";
 40 INPUT Q$
 50 IF Q$ = "Y" THEN 70
 60 GOSUB 1000
 70 GOSUB 9000
 80 PRINT
 90 GOSUB 2000
100 GOSUB 3000
110 GOSUB 4000
120 GOSUB 5000
130 GOSUB 6000
140 IF X > 1 THEN 110
150 PRINT "DO IT AGAIN (Y OR N)"
160 INPUT Q$
170 IF Q$ = "Y" THEN 210
180 PRINT "SO LONG GANG"
190 PRINT "THE END ..."
200 END
210 PRINT "SAME PLAYERS (Y OR N)";
220 INPUT Q$
230 IF Q$ = "Y" THEN 100
240 GOTO 80
1000 PRINT "TWO PLAYERS TAKE ROCKS FROM A"
1010 PRINT "PILE.  THE ONE STUCK WITH THE"
1020 PRINT "LAST ROCK LOSES."
1030 PRINT
1040 PRINT "YOU CAN'T TAKE MORE ROCKS IN"
1050 PRINT "YOUR TURN THAN ARE LEFT IN"
1060 PRINT "THE TOP ROW."
1070 PRINT
1080 PRINT "HERE WE GO ...."
1090 RETURN
1999 REM  "TYPE NAMES"
2000 PRINT "TWO PLAYER'S NAMES:"
2010 PRINT "#1";
2020 INPUT P1$
2030 PRINT "#2";
2040 INPUT P2$
```

```
2050 PRINT "OK (Y OR N)";
2055 INPUT Q$
2060 IF Q$ = "Y" THEN 2080
2070 IF Q$ = "N" THEN 2000
2080 IF P1$ = " " THEN 2000
2090 IF P2$ = " " THEN 2000
2100 IF P1$ = P2$ THEN 2000
2110 LET P = INT(10*RND(1))
2120 IF P => 5 THEN 2150
2130 LET P = 1
2140 RETURN
2150 LET P = 2
2160 RETURN
2999 REM  "PILE UP THE ROCKS"
3000 FOR I = 1 TO 9
3010 LET R(I) = I
3020 NEXT I
3030 RETURN
3999 REM  "PRINT LAPIDES"
4000 FOR I = 1 TO 9
4005 IF R(I) = 0 THEN 4050
4010 FOR J = 1 TO R(I)
4020 PRINT "*";
4030 NEXT J
4040 PRINT
4050 NEXT I
4060 RETURN
4999 REM  "PLAYER INPUT"
5000 IF P = 1 THEN 5040
5010 LET P = 1
5020 PRINT P1$"'S TURN"
5030 GOTO 5060
5040 PRINT P2$"'S TURN"
5050 LET P = 2
5060 INPUT Q
5070 IF Q < 1 THEN 5090
5080 GOTO 5110
5090 PRINT "PLAY FAIR!"
5100 GOTO 5060
5110 FOR I = 1 TO 9
5120 IF R(I) <> 0 THEN 5140
5130 NEXT I
```

```
5140 IF Q =< R(I) THEN 5170
5150 PRINT "ONLY"R(I)"POSSIBLE"
5160 GOTO 5090
5170 LET R(I) = R(I)-Q
5180 RETURN
5999 REM  "CHECK FOR WIN"
6000 LET X = 0
6010 FOR I = 1 TO 9
6020 LET X = X+R(I)
6030 NEXT I
6040 IF X > 1 THEN 6080
6045 IF X = 0 THEN 6110
6050 PRINT "HA HA, ";
6060 IF P = 1 THEN 6090
6070 PRINT P2$ " WINS."
6080 RETURN
6090 PRINT P1$ " WINS."
6100 RETURN
6110 PRINT "DUMMY!"
6120 PRINT "YOU TOOK THE LAST ONE."
6130 RETURN
9000 REM  "RANDOM NUMBER ROUTINE"
9010 LET Z = RND(1)
9020 RETURN
```

M



Pick up on the gaming theme of Dice, devise an internal card-deck management scheme, and . . . *presto*! We have another game for our library. For the sake of variety, however, there are some other differences. They are noticeable in the instructions as printed by the program itself.

```
2, 3, OR 4 PLAYERS
DRAW CARDS IN TURN.
YOU MAY DRAW FROM 2
TO 9 CARDS PER TURN
(OR YOU MAY PASS).
SCORE 1 POINT FOR EACH
CARD DRAWN -- UNLESS
YOU GET A PAIR. ANY
MATCH ENDS YOUR TURN
AND YOU GET NO SCORE.
GAME IS 20 POINTS -- FIRST
ONE THERE WINS.
```

In the referenced game of Dice, a turn required a decision by the player for each roll. A turn in *Match* consists of but one operator entry: the number of cards to draw. According to the instructions a valid entry can be zero (to pass) or any digit from two through nine.

There is a reason for the different playing conventions. With dice the odds for what may be rolled in any one turn are the same as for every other turn. With a deck of cards, however, the odds

change with every draw in any card game where the deck is continually being used up. This program does use a shrinking deck, but when it is exhausted a new one is automatically brought in to play. There are some possibilities for using one's skills in this case rather than depending entirely on Lady Luck.

As each card is dealt a parenthetical note is output also, which shows the draw number (ranging from one to fifty-two, serially). Suppose then, the last card dealt was number 47. There are five cards left in the deck. If your infallible photographic memory says there are no pairs remaining in the deck an entry of five is strategically sound.

What about taking a chance with a draw request for six? In the example just cited the sixth card will introduce a newly shuffled deck. Only Lady Luck knows what the first card from a new deck is apt to be. (Drawing any pair does mean you'll get no score for the turn, remember.)

The *Match* program does use an honest deck (though you may have to use this text, a copy of your program's listing, and maybe even pencil and paper to prove it to some skeptics). One of the design goals for this program was to devise an authentic shuffling mechanism that cannot be faulted by any arguments concerning mechanical bias. During your study of what follows you too may believe that here we have an honest game.

MATCH FROM THE OUTSIDE

One picture is worth a thousand words. To see the program, then, look at Fig. M-1. The several parts of Match are shown pictorially by the program template, although as we all know there are no blocks around the statements in the computer's memory.

If you are not the sort of person that begins reading a book at midpoint you will agree there is nothing new about the top two blocks. Out of the program's initialization area there is an optional call to a subroutine to print the game's instructions. Ho hum. Then a game is set up.

The start game section of the program does brief housekeeping and performs a jump to the new deck module. The arrangement of the blocks and the connecting lines infer that the 2000-series coding (lines 2000 to 2150) is a shared subroutine. It is called upon just before a game starts, and thereafter it may also be jumped to whenever an attempt is made to draw cards (lines 3000 to 3150) from an empty deck.

The terraced structure to the right of the mainline, with but one path out and back, predicts two things. The architecture of *Match* is

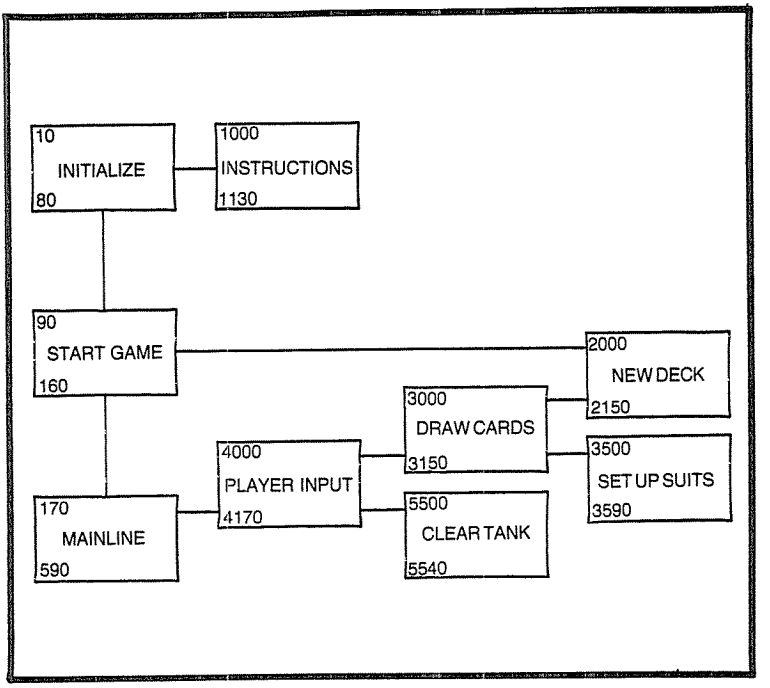


Fig. M-1. Program template for Match.

characterized by nested subroutines, and the mainline must condition the commonly used variables prior to each repeated trip up the chain. Two other inferences are possible from this picture as well.

The clear tank block (lines 5500 to 5540) can be assumed to mean that a table is used to hold all of the cards for a given draw. That is true. So is the implied case that card values and their respective suits are managed apart. That is why there is the block labeled setup suits. This design was laid out after the deck structure was conceived, and that is probably the best order for describing it all. Skeptics may begin taking notes now.

The Dubitable Deck

Needed: a table. There are fifty-two cards in a regular playing deck, so the table ought to have fifty-two fields. It does. Figure M-2 shows an abbreviated picture of the program's *table D* and its initially coded contents.

Creation of a new deck is not difficult at all. Two loops are used by the program to generate numerical codes to represent all of the cards, including their suits. One of the loops counts from one to thirteen and the other counts from zero to three. The second

counter is the one used for the suits; it is incremented by one each time the first counter passes thirteen. (The one to thirteen counter is reset to one at the same time.)

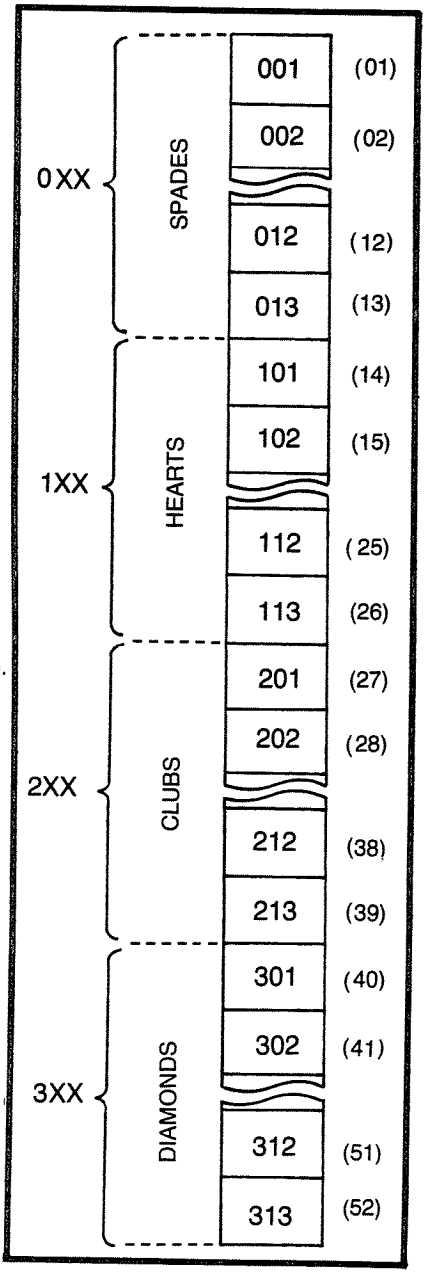


Fig. M-2. Table D, the deck in Match.

The table is loaded from the top downward, a field at a time. Each code stored there is a combination of the two counters. The zero to three count becomes the code's high-order digit. The other counter (one to thirteen) is formed as the two digits to the right of the suit code. Once the table is fully loaded all that remains is to shuffle the deck. That is phase 2 of the new deck process.

The Shuffle

A programming loop is used for the shuffling task. Naturally. To insure that the job is thoroughly done, the loop executes fifty-two times.

With each pass through the loop the steadily incrementing counter is used as one of a pair of table subscripts. The other subscript is obtained by a technique that depends on the RND function. Near the end of the loop, using the pair of subscripts for accessing two of the table fields, the codes at those locations are swapped. Because the one table reference is serially incremented, and because the loop runs for the length of the deck, every card has the potential of being shuffled with some other. It is possible, of course, that at any given point in time the two subscripts may be alike, resulting in a self-swap.

This is believed to be a natural consequence, equally possible if a real deck of cards was shuffled by hand. So too is the possibility that as the task proceeds, a card may be shuffled off to some random location and in a later pass get swapped back to its original spot. In some ways the whole of this process is more thorough than some lazy card players.

Some could argue that it is unnatural to riffle the cards for an exact number of times. For that matter they could also argue about the fact that each new deck begins in a predictable sequence. So what? Some people would argue with the devil, too. They are not likely to be impressed with the way RND is implemented either.

Using a standard form of expression—let J=INT(100*RND (1))—it is easy to ask your system to hand up a random value in the 00 to 99 range. The dimensions of *table D* dictate the need for a subscript of from one to fifty-two, inclusive. In this program the algorithm expressed in BASIC chooses to refuse a zero and insists on repeated function calls until something more significant is offered.

Rather than risk infuriating your operating software by being too obstinate, if the RND call fetches a number larger than fifty-two we will use it. How? By simply cutting it in half (divide by two and truncate the 0.5 for the oddballs).

The result of dividing any number of the series from 53 to 99 by two and ignoring the remainder will be some number from 26 to 45. To force the value over the table's halfway mark the arithmetic expression used in this implementation includes the addition of a one to whatever is derived by the division step. This seems to be fair. The likelihood that an RND call will fetch a number from the set one to fifty-two is slightly skewed—the odds are a little better than fifty-fifty that the number will be usable just as it is first offered.

The astute may observe that, conversely, nearly half the time the division trick is resorted to. This does mean that considerably more than half the time the second of the two subscripts that are used is aimed into the lower portion of the deck. You might counter by recalling that exactly half the time the shuffle loop's counter is addressing the top area of the table.

If you are still suffering a doubting Thomas, here is another idea: ask him or her to do a careful audit on the movement of the cards in a real deck as they are shuffled by hand. This might not eliminate your friend's eristic voice, but it should buy you enough time to study the rest of what is inside the *Match* program.

MATCH FROM THE INSIDE

Skipping quickly over the rote statements in the listing notice the table allocations in line 80. There is the D-for-52. The deck table has already been intimately described. The *M$* and *U* assignments that are there also deserve a brief exposure. These are, respectively, the author's nicknames for the match table and the user table. We'll delay their intimacies until they become involved.

The DIM in line 80 is the last function of the program's initialization. From line 90 down through line 160, all of which is within the startup of a game, are nine lines of dialog to *a*scertain how many players are to be *a*llowed. When *a*sked for, the *a*nswer must be *a*bsolute: two, three, or four. It is stored in *a* variable that is *a*rbitrarily called *A*.

After all of that the GOSUB 2000 in line 170 sets up the first deck of cards. The previous description did not include the variables used during the card shuffling task. So it is time for them to be introduced.

The two loop counters are *I* and *J*, as seen for the first time in lines 2000 and 2010. The left half of the LET expression in line 2020 uses both of the variables in conjunction as a serial subscript for addressing *table D*. On the right, *I* times 100 plus *J* will create a three-digit number with the suit code in the high-order position. The

loop-and-store phase continues for fifty-two times, finally falling through to statement 2050.

The shuffle phase extends down to line 2130. Temporary worker *S* is used by this task during the swapping of the card codes. Later, in another routine, *S* is again used for transient purposes (meaning then, suit). More about that later.

Finishing up with the card shuffling module, the simple variable *D* is set to zero in line 2140 just before the RETURN is executed. Use of this counter is exclusive—it tells the number of the last card dealt. When it reads fifty-two the deck is depleted. But now, with a new deck in place, let us return to the mainline at statement 180.

Here we find the ubiquitous *U*. There are really two of them: one is a simple nonsubscript variable, and the other is a table name. They both have to do with the program's users. The single field of *U* is a counter. It is initially set to one in line 180; and whenever it is incremented beyond the value in *A* (checked by line 400), it is set back to one in line 410. The *table* called *U* has four counters, one each for the four possible players. These fields are for storing the players' scores.

Both types of *U* are displayed by the PRINT expression in line 420. This serves as part of the input prompt: it shows which player is supposed to be up, and in parentheses displays that player's current score. After the player takes his turn his total is tested to see whether he has attained or has passed the game's twenty-point limit. The player's score is updated by the number of cards that are drawn; in the event of a match, his *U* score remains intact. The scorekeeping is done within the input module, as called for by the GOSUB 4000 in statement 430. That is the area to study next.

The rest of the input prompt consists of the one word: DRAW. As soon as this is printed by line 4000 a quick jump is taken to line 5500. Most likely, before the player can respond to the input request, the loop from line 5500 to line 5540 will be completed. That task will clear the nine-element tank that will hold the individual cards as they are drawn. (Notice that *Q2* serves as the loop and the subscript variable in that task.) The final function that is accomplished before the subroutine recovers to line 4020 is the loading of *Q* with 99.

Program execution is suspended on execution of the INPUT in line 4020. A default entry (nothing typed but the terminating key) will function as if the player had typed 99 due to the earlier preconditioning. This convention, plus the quality checking that is done from line 4025 to line 4050, will keep the player from making careless keystrokes.

Recall the prediction about nested structures. Here they are. There are two concentric loops between lines 4060 and 4094. The outside loop uses *Q1* as its counter. This is the loop that is controlling the draw. That is why it is limited to *Q* times. (*Q* has the player's draw request in it.)

The inside loop extends from line 4080 to 4086. As each card is drawn the *Q2* counter runs the loop for nine times, unless the search finds a match or a blank spot in *table M$*. A blank spot in the table means this is a good draw; so statement 4090 saves this card right there. The *Q2* stops then, and the first loop is executed again to draw another card.

If another draw results in a match with one already in the table the conditional in line 4084 will branch down to line 4110. The player's request in *Q* is zapped in line 4110 (no score allowed), the machine chortles because of it, and the RETURN goes back to the mainline.

Other nested loops are possible, also. One of them happens with each draw because of the GOSUB 3000 in line 4070. This is also the start of a nested subroutine jump sequence.

At the top of the draw cards module (in line 3020) is another possible jump. This one goes to the new deck procedures which, as has already been covered, has two more loops. The conditional in line 3000 only lets this task be executed after a deck is exhausted; otherwise, the D-for-draw number is incremented in line 3030 and a card is taken from the top of the deck.

The number of the card to be picked up is displayed by line 3040, and the code from the table at the *D* location is moved to C-for-card. Time out now for a short subroutine jump.

The DATA definitions in lines 3060 and 3070 are used by the task that begins in line 3500. That is why they immediately precede the GOSUB in line 3080.

Two things are accomplished between line 3500 and the RETURN in line 3590. The card code (in *C*) is parsed into two numbers by lines 3500 and 3510. Now the *C* variable only has the card's ordinal number; the high-order digit is now in *S*. The defaulting type of logic from line 3520 to 3590 will wind up with *S$* loaded with the printable name of a suit. That is all that this task does; but remember that the *C* code is now a whole number of from one to thirteen. Back to line 3090.

Here we have a READ loop. The FOR-NEXT setup from line 3090 to 3120 uses the one to thirteen portion of the original card code to halt the READ task with the right DATA name in *C$*. The

RESTORE in line 3130 preconditions the DATA pointer for use the next time.

After all of that the PRINT expression in line 3140 to display the card does seem almost anticlimactic. It is. The *return* in line 3150 starts the relinking process, going back to line 4080. From there the RETURN in line 4130 will eventually get back to the mainline.

It is recognized that the nesting of both loops and subroutines may seem confusing. Mostly just to people, though. Try coding and loading the *Match* program as presented here. Chances are your computer won't be confounded. And the chances are equally good that you and your skeptical friends will enjoy this game.

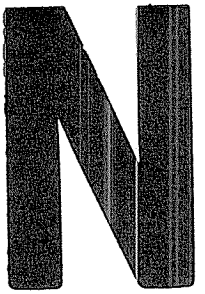
THE PROGRAM

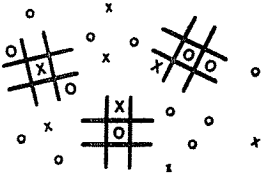
```
 10 REM  "MATCH"
 20 REM
 30 GOSUB 9000
 40 PRINT "KNOW THE RULES (Y OR N)";
 50 INPUT Q$
 60 IF Q$ = "Y" THEN 80
 70 GOSUB 1000
 80 DIM D(52), M$(9), U(4)
 90 PRINT "HOW MANY PLAYERS (2-4)";
100 INPUT A
105 LET A = INT(ABS(A))
110 IF A < 5 THEN 140
120 PRINT "TOO MANY"
130 GOTO 90
140 IF A > 1 THEN 170
150 PRINT "TOO FEW"
160 GOTO 90
170 GOSUB 2000
180 LET U = 1
190 PRINT
200 FOR I = 1 TO 4
210 LET T(I) = 0
220 NEXT I
400 IF U <= A THEN 420
410 LET U = 1
420 PRINT "PLAYER #"U"("U(U)")";
430 GOSUB 4000
440 IF U(U) => 20 THEN 480
450 LET U = U+1
```

```
 460 PRINT
 470 GOTO 400
 480 PRINT "CONGRATULATIONS"
 490 PRINT
 500 PRINT "ANOTHER GAME (Y OR N)";
 510 INPUT Q$
 520 IF Q$ = "Y" THEN 560
 530 PRINT "GOOD - I'M TIRED."
 540 END
 560 FOR I = 1 TO 4
 570 LET U(I) = 0
 580 NEXT I
 590 GOTO 90
 999 REM  "INSTRUCTIONS"
1000 PRINT
1010 PRINT "2, 3, OR 4 PLAYERS."
1020 PRINT "DRAW CARDS IN TURN."
1030 PRINT "YOU MAY DRAW FROM 2"
1040 PRINT "  TO 9 CARDS PER TURN,"
1045 PRINT "  (OR YOU MAY PASS)."
1050 PRINT "SCORE 1 POINT FOR EACH"
1060 PRINT "  CARD DRAWN -- UNLESS"
1070 PRINT "  YOU GET A PAIR.  ANY"
1080 PRINT "  MATCH ENDS YOUR TURN"
1090 PRINT "  AND YOU GET NO SCORE."
1100 PRINT "GAME IS 20 POINTS -- FIRST"
1110 PRINT "  ONE THERE WINS."
1120 PRINT
1130 RETURN
1999 REM  "CREATE CARD DECK"
2000 FOR I = 0 TO 3
2010 FOR J = 1 TO 13
2020 LET D(J+I*13) = J+I*100
2030 NEXT J
2040 NEXT I
2050 FOR I = 1 TO 52
2060 LET J = INT(100*RND(1))
2070 IF J < 1 THEN 2060
2080 IF J < 53 THEN 2100
2090 LET J = INT(J/2)+1
2100 LET S = D(J)
2110 LET D(J) = D(I)
2120 LET D(I) = S
```

```
2130 NEXT I
2140 LET D = 0
2150 RETURN
2999 REM  "DRAW CARDS"
3000 IF D < 52 THEN 3030
3010 PRINT "NEW DECK"
3020 GOSUB 2000
3040 LET D = D+1
3045 PRINT "DRAW #"D;
3050 LET C = D(D)
3060 DATA ACE, 2, 3, 4, 5, 6, 7, 8, 9, 10
3070 DATA JACK, QUEEN, KING
3080 GOSUB 3500
3090 FOR I = 1 TO C
3100 READ C$
3110 IF I = C THEN 3130
3120 NEXT I
3130 RESTORE
3140 PRINT TAB(12) C$; S$
3150 RETURN
3500 S = INT(C/100)
3510 C = C - S*100
3520 LET S$ = " HEARTS"
3530 IF S = 1 THEN 3590
3540 LET S$ = " CLUBS"
3550 IF S = 2 THEN 3590
3560 LET S$ = " DIAMONDS"
3570 IF S = 3 THEN 3590
3580 LET S$ = " SPADES"
3590 RETURN
3999 REM  "PLAYER INPUT"
4000 PRINT " DRAW";
4010 GOSUB 5500
4020 INPUT Q
4025 LET Q = INT(ABS(Q))
4030 IF Q = 0 THEN 4130
4040 IF Q > 9 THEN 4140
4050 IF Q < 2 THEN 4140
4060 FOR Q1 = 1 TO Q
4070 GOSUB 3000
4080 FOR Q2 = 1 TO 9
4082 IF M$(Q2) = " " THEN 4090
4084 IF M$(Q2) = C$ THEN 4110
```

```
4086 NEXT Q2
4088 GOTO 4094
4090 LET M$(Q2) = C$
4094 NEXT Q1
4100 GOTO 4160
4110 LET Q = 0
4120 PRINT "MATCH - HA HA!"
4130 RETURN
4140 PRINT "ILLEGAL"
4150 GOTO 4020
4160 U(U) = U(U)+Q
4170 RETURN
5499 REM  "CLEAR TANK"
5500 FOR Q2 = 1 TO 9
5510 LET M$(Q2) = " "
5520 NEXT Q2
5530 LET Q = 0
5540 RETURN
9000 REM  "RANDOM NUMBER ROUTINE"
9010 LET Z = RND(1)
9020 RETURN
```





Naughts & Crosses

On the whole this book is meant to entertain. But not without class. *Naughts & Crosses* does seem more eloquent than tic-tac-toe. Besides, the letter *T* had already been committed.

By whatever name this game should be recognizable to children the world over; and your own young friends will undoubtedly enjoy being able to "beat a computer." Most every youngster at about the third grade or so knows the elementary strategies for this game. That is about the age at which they should have an advantage over the educational level of this program. Yet, it is amusing—especially to build it and to get it to work properly.

That is the true ambition of this example, anyway. Here are the mechanics and structures for programming tic-tac-toe games. The more adventuresome can go from here to devise more complex offensive internals. Some may even wish to go all the way and attempt a three-dimensional version. There is at least a foundation here for such designs, as well as some academically interesting techniques.

A DESIGN APPROACH

There is first the matter of the playing board. Secondly, a means is necessary for the player to indicate a move choice. Fancy graphic output could be done, even with a data printer or display; but is it all really needed? This program supposes not. Here is a sample output.

1 2 3 O • •

4 5 6 • X •

7 8 9 X • O

Three lines are always printed; the numbers on the left are data constants, and the period characters are replaced as a game pro-

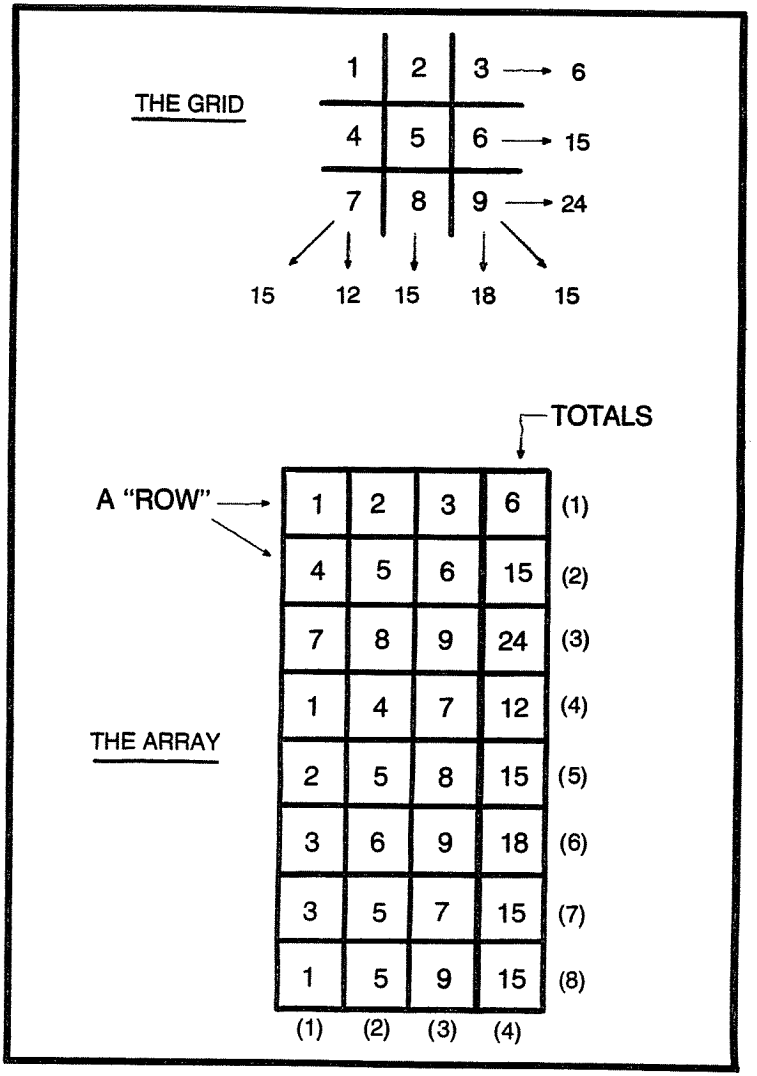


Fig. N-1. Row values used in table R.

ceeds. The traditional lines that are handy when playing with a pencil are omitted, but they are easily imagined. The periods are there to enable the eye to see where the lines ought to be.

There is also a perceptual advantage to the arrangement of the numbers. More importantly, the numbers provide a mechanism for the player to indicate, through the keyboard, where he or she wishes to place his or her marker. The markers themselves are built into the program. The computer always owns the *O*, and *X* is exclusively the human's.

Otherwise, in most ways, the usual mechanics for managing a two-player game—one fleshy and one phony—must be programmed for. There are two tasks conspicuously absent, however. There are no scorekeeping routines, nor is there any real need for a random-number generating function. The rudimentary strategy that the program uses is finite, although some variety could be imparted to the computer's rote selections if an RND was used. In which case, if the computer could find no move dictated by logic, it could choose haphazardly. Even though this program never guesses it is surprising how few players ever detect the machine's offensive pattern.

For the most part this program does assume it is always on the defensive. He who gets to go first alternates, game to game, but the computer may lose this advantage quickly. The built-in intelligence will capture the center position, if possible, block any third chance for the player (when obvious), and it will never miss an obvious move to win itself.

Otherwise, an open move is achieved by picking from a table of numbers. In fact, the whole of this program is table driven. There is a table for knowing the player's moves, one for holding the machine's markers, and another that represents both. There is also a master table that is made up of numeric constants for determining position vacancies. Because so much depends on these structures and their uses, that is what is described next.

THE TABLES IN NAUGHTS & CROSSES

A tricky task in tic-tac-toe games is knowing when a win has occurred or is imminent. There are eight rows possible: three vertical, three across, and the two diagonals through the center. The technique used here includes assigning a number value to each position to use in doing simple row addition for the positions that are taken up. By knowing in advance what the maximum for a row can be, if any position is not taken, the temporary sum will be less than is possible for that row.

To better show some of the contents possible in *tables H, C,* and *B,* Fig. N-2 illustrates a partially played game. Supposing the computer was about to make an offensive play: notice the last group of fields in *table B* (elements 7, 8, and 9). Added together they equal exactly twenty. The only way a row can add up to twenty (in *table B*) is if the computer has marked twice in a row and the third spot is

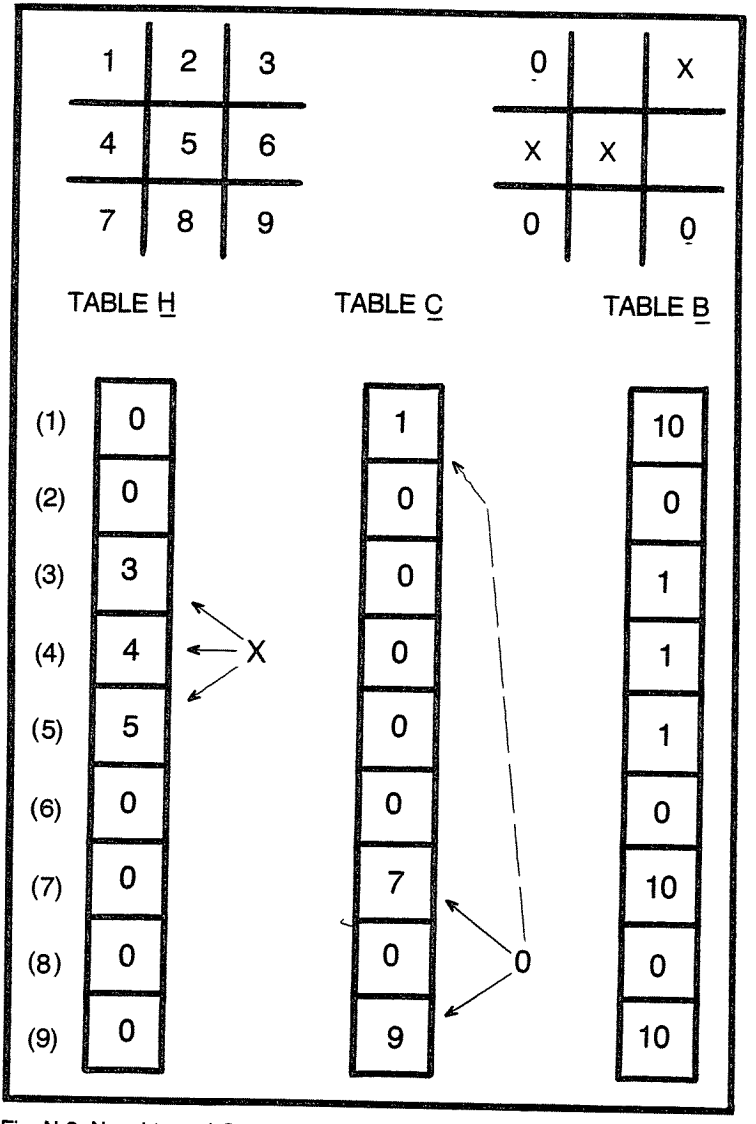


Fig. N-2. Naughts and Crosses, tables H, C, and B.

empty. By adding together the corresponding fields in *table C*—which is sixteen—and by subtracting that amount from the appropriate total in *table R* (twenty-four), the number eight is deduced.

A similar scheme works for a defensive play by the machine. Fields 4 and 5 of *table B* add up to two—no more and no less. Any row that totals two exactly in this table means the machine had better make a blocking move pronto. Again, using arithmetic on the corresponding fields in the human's table (and the total from *R*), the position number for the blocking play can be easily determined.

The reason for using ones and tens in *table B* is revealed in the following list of numbers. These are the only totals possible for any row.

0 1 2 3 10 11 12 20 21 30

Of these, the important ones are two, three, twenty, and thirty. Both two and twenty indicate disaster is imminent, depending on whose side you're on. In the event any row adds up to three or thirty the jig is up; in the first case the computer lost, and in the second it won.

Overall, *table R* is the master table that drives the program. The other tables are storage tanks. Whenever it is necessary to scan the board the subscript constants are used from the master table. What happens internally? A loop is executed eight times, each time using one of the rows of numbers from this table. The numbers themselves are used as individual addressing offsets into each of the storage tables.

When a game begins all three of these tables must be cleared. From there on they are posted on the fly. Any move by either player requires two postings. *Table H* or *table C* is marked, depending on whose play is being recorded; and *table B* is posted at the same time with either a one or a ten.

The way these things are accomplished will be seen in the study of the program's contents. There are some interesting coding techniques shown there, also—especially some of the compound multidimensioned, multiple subscripting expressions used. Before delving into that delightful detail, however, a few comments are appropriate on how *Naughts & Crosses* is laid out.

MODULES, MAPPING & MORE

The program template shown in Fig. N-3 shows the conceptual organization of *Naughts & Crosses*. My editor's sensitivities were

aroused when I implied this drawing is a modules map. "Map? It looks more like an atlas!"

A few minutes' study should show, though, that the architecture of this program is not all that complex. As usual at the top are the housekeeping routines. The broken-line box is the mainline of the program. Internally it has a pair of paths—down the left side for the human, down the right for the computer. Further to the left are supporting subroutines that are proprietary to people and such, and

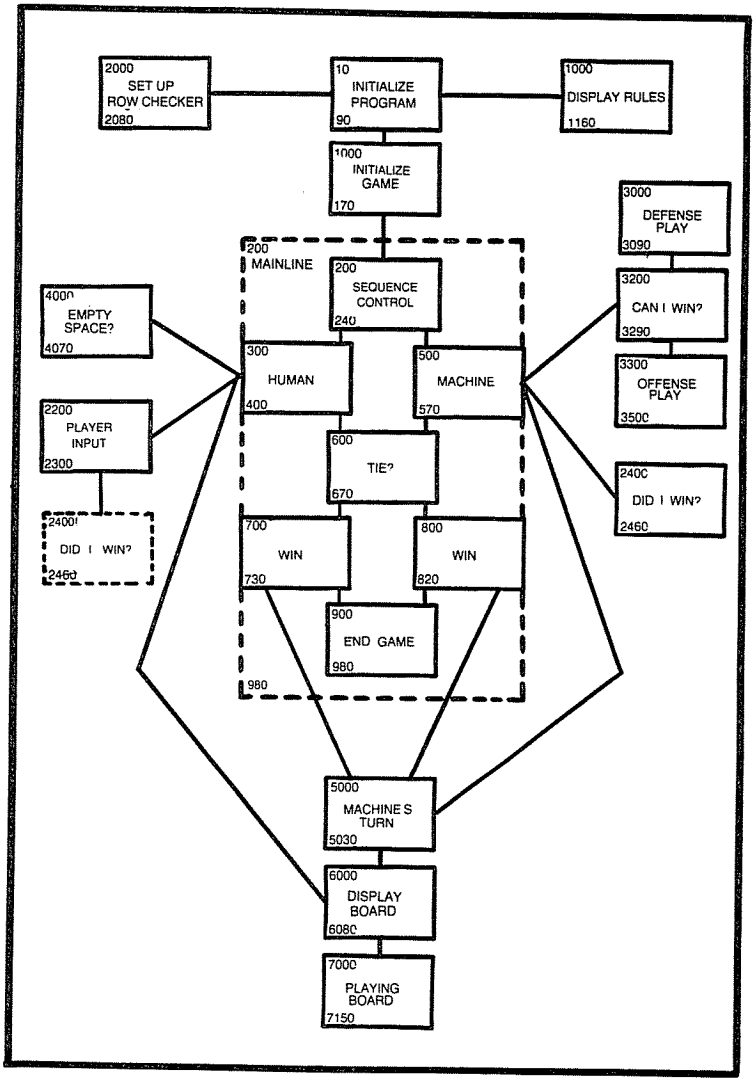


Fig. N-3. Program template for Naughts and Crosses.

on the far right are those routines that do the mechanized opponent's thinking. Notice also there is a set of subroutines at the bottom that are shared, from two different points, by both factions.

The illustration shows the gist of the design. To learn how the program works, our study continues with a look at the listing.

HOUSEKEEPING & MAINLINE

The first statements of significance are those at lines 30 and 40. These establish explicitly the program's mass storage areas. The DIM in line 30 defines *R* to be an array. Conceptually, the left-hand parameter of eight specifies depth; the four to the right of the comma indicates width. *Tables H, C,* and *B* are all single-dimensioned in line 40, each with a depth of nine. These are one-shot initializing statements. So is the next.

The GOSUB 2000 is unconditionally executed to embrace the task that loads the row table (the eight by four array) with the numeric constants. Once the array is fully conditioned this exercise is not again needed. The values are read many times over, but these constants remain unaltered throughout the entire program's execution. Because it is only done once, a quick glance is appropriate here to see how it is done.

The DATA statements in lines 2010, 2020, and 2030 are defining, for logical purposes, a continuous string of twenty-four digits. The FOR-NEXT loop that extends from line 2040 to line 2070 grabs the numbers as sets of three and moves them to the row-checker array. The READ expression in line 2050 loads the constants directly into the table. Notice that as the *I* variable is incremented the table is filled a row at a time from the top. Concurrently, the fourth column of the array is compiled by the addition expression in statement 2060. Eight times through, and back to the mainline.

The balance of the program's initializing logic extends from line 60 to line 90. This includes nothing more than the option to print a brief description of the game. If the operator is being coy and answers with an *N*—meaning no, I don't know how to play—the rules module is invoked. Enough of that. Most programmers know how to play tic-tac-toe, and most kids can figure out the subroutine from line 1000 to 1160. The RETURN takes us back to the mainline at statement 100.

Time out now for a nine-times loop. Zeros are moved into all of the fields in the storage tables by the FOR-NEXT task that spans from line 110 to 150. When the *I* counter reaches nine a fallthrough

is automatic. The last two tasks for initializing a game are in statements 160 and 170. The *Q* and *W* workers (query and win) are used in several places as we shall soon see.

Sequence Control

The *P* variable has but one purpose: the players are identified internally as number *1* and number *2*. The superiority of mankind is acknowledged by making the machine second. For the first game only the expression in line 230 sets *P* to one. From then on player sequencing is continuously automatic, alternating from one to two and back, even across the end-of-game boundary.

Human's Play

In line 310 *P* is switched to two. This is how the alternation is accomplished. It is reset to one immediately upon entry into the machine's module. From line 320 to line 400 is the rest of the control block for people procedures. A jump is made to line 6000 to display the playing field; then a quick test is made to see whether there are any open spaces in the grid. Notice that *Q* is turned on with a one in line 320. When a relink after the GOSUB 4000 occurs, if *Q* was switched off with a zero, the grid is full. Because a win trap had never taken place from anywhere else, a tie is implied; so the conditional in line 350 will exit to the draw task at line 600. Otherwise, if *Q* is returned with any value in it, our human friend gets to play.

Again, *Q* and *W* are both turned off in lines 360 and 370, and a GOSUB 2200 is executed. There the player gets to type his or her position choice, which is then qualified as to whether it is legal; the program then returns to line 390. If *W* comes back with a three in it the human won and a branch to line 700 will make him or her the victor. Any value in *W* other than three means the computer gets a chance; so the branch in line 400 goes to line 200 for resequencing the mainline flow.

Computer's Play

Coding from line 500 to line 570 is logically similar to that for the human's control block. A series of jumps (lines 5000, 3200, and 2400) will print the board again, then fetch the machine's move, and check if it lucked out and won. Notice that in line 550 if *W* comes back as thirty a branch is taken to announce the computer's victory. If *Q* was set to off the game should be ended as a draw. This logic is the same as for when the human was in command. Any residue in *Q*

means there is more to come; so the branch in line 570 loops back to the top of the mainline.

Tie, Win, Or End

The series of code from line 600 down to line 900 are brief tasks, mostly to announce the outcome and to recondition the *P* variable whenever necessary. The end-of-game wrapup is from line 900 to line 980. When asked to play again either a branch to line 100 will occur (from line 930) or the program is ended with a complimentary closing. Assuming you wish to continue it's time to look at the supporting modules, beginning with those for human opponents.

Empty Space Test

The module from line 4000 to line 4070 is used preparatory to the human's play to see if there is room for his or her marker. All that is necessary to learn this is a quick check of *table B*. Any zero found there will interrupt the search task with an exit via line 4060. The loop counter is moved into *Q* only as a nonzero indicator. If the loop runs a full nine times the tank is full (thus, so is the grid); so the default exit is taken (which will cause *Q* to be returned empty).

Player Input

There is a prompt in line 2210, an INPUT in line 2220, and an editing mask in line 2215. This last was a later addition to clean up sloppy typing. The INT and ABS functions working together will force *Q* to be a clean integer, ignoring junk such as decimal digits or a minus sign. The conditionals in lines 2230 and 2240 insure that what is left is between one and nine, inclusive. Otherwise, the player is admonished and forced to try again. Quality control over solicited input is not yet complete. The attempted move must be further checked against the existing state of the board.

Using the contents of *Q* as a subscript statements 2270 and 2280 look at both the human's and the computer's tanks to see if there is already a marker there. If so an illegal move is being attempted; so the previous error route is taken. If the move is valid, programmatically at least, the *Q* number is moved into *table H* at the *Q* location, and *table B* is posted at the *Q* address also. This routine then bottoms out right into the win checker.

Check For Winner

Coding from line 2400 to line 2460 makes use of the subscripting constants in *table R* to check the combinations of totals in *table B*.

Any row that adds up to either three or thirty will cause this loop to exit instantly with the win indicator intact. Anything else in *W* will mean the loop ran a full eight times. The residual result is meaningless.

The previous routine is a shared module, used by both players, and that is why both the three and the thirty are qualified. There is another set of mutually shared code, and this is a convenient time to look at it.

Display Board

Three print lines are needed, with constants on the left and a picture of the board's contents on the right. The series of statements from line 6010 to line 6070 contains number grid constants within PRINT constructs plus alternating jumps to a subroutine to fetch up the board a line at a time. The first jump is to line 7000, which conditions a *from* and a *limit* variable (*F* and *L*). The next two lines are printed by using a jump into line 7020 so that the FOR-NEXT parameters may be built upon. Conditional printing logic from line 7050 to line 7100 will output a cross (x) a naught (o), or a period, depending on the contents of the three tables that reflect the playing board's status.

All that remains of our *Naughts* program is to look at how the machine is able to take its turn. For this there are three closely coupled tasks, all coded in the 3000-series line numbers. The order of processing first determines whether the computer can win or a defensive play is needed. If neither is the case the logic assumes the initiative and the computer is permitted a bold attacking play—well, sort of bold, anyway.

Check For Win Move

A short looping sequence from line 3210 to line 3240 will find any row in *table B* that adds up to twenty. Although this may not happen often, when it does, zap: the branch to line 3260 takes over, the winning move is chalked up, and the victory will be discovered back in the mainline. More frequently, the loop will exhaust the limit of eight and a branch to line 3000 will take place.

Check For Defense

The looping scheme from lines 3000 to 3040 is just like the previous one—this time it's a two that is being looked for. If found the indication is that there are two ones codified in a row in *table B*; hence, the opposition must be blocked. The branch to line 3060 will

enter the logical sequence that argues which position is the vacant one, based on the depth that *I* indicates the scan was made to. In line 3060 we see the arithmetic that subtracts the value of the marked locations from the total possible for that same row. The result is then placed into *Q* and becomes the move choice for the computer.

If the defense-testing logic executes the full eight times without finding a pair of the opponent's markers in one row (with a blank spot) the logic switches to the offensive. The GOTO 3300 in line 3050 is the way the program gets there.

Make Offense Play

Right off of the bat, is the center spot taken? This question is asked by the expression in line 3310. If vacant *Q* is loaded with this strategically valuable number (a five, and a hasty exit is taken by means of the branch in line 3330. Now for the hard part. If the center spot has already been burned all of the remaining vacancies should be thoroughly analyzed to determine the most strategically sound play. That is what should be done. What this program does is much easier. The loop that runs from line 3340 to line 3360 just looks down through *table B* for an empty spot. Ho hum.

If you are inclined to do so this is a good place to begin trying to raise your computer's IQ. At least with *Naughts* your experimenting won't have to include teaching it how to play elementary tic-tac-toe.

THE PROGRAM

The master table is labeled with an *R* (for row), and it is set up as an array. It is four columns wide and eight rows deep. The values that are stored in this table and how they are derived are shown in Fig. N-1. There are eight win rows possible in the three-by three grid. Each of the numbers, per row, are stored in the first three table elements, per row. The fourth column of the table holds the sum of the numbers in each row. Again, *table R* is generated once, to be used thereafter as a source of addressing subscripts and total-testing constants.

There is a table, also, for each player. One is tagged with an *H* (for human), and the other is called *C* (for computer, naturally). Both of these tables are single-dimensioned and both have nine fields, one for each of the possible marker locations on the playing grid. In the beginning both tables are empty. Whenever a turn is taken, amusingly enough, the number denoting a move is itself stored in the player's table; at the spot corresponding to the move.

Suppose the kid on the keyboard had played a one and a three. The first three fields in *table H* would read out as 1-0-3. Added

together, that's four. Four from six (looking at the fourth element of the first row in *table R*) is two. The difference of two not only indicates a vacancy in the top row of the grid, it is the position value of the vacant spot.

Because each player has an individual table there is a need for some cross checking. Rather than having to continuously look at both tables—depending on who is up at the time a play takes place—by either, a third table is posted to indicate all of the markers thus far played. It benefits both, so it is called *B*. (Or maybe it was B for Board—whatever you like.)

```
 10 REM  "NAUGHTS"
 20 REM
 30 DIM R(8,4)
 40 DIM H(9), C(9), B(9)
 50 GOSUB 2000
 60 PRINT "KNOW HOW TO PLAY (Y OR N)";
 70 INPUT Q$
 80 IF Q$ = "Y" THEN 100
 90 GOSUB 1000
100 REM "INITIALIZE PLAYING BOARDS"
110 FOR I = 1 TO 9
120 LET H(I) = 0
130 LET C(I) = 0
140 LET B(I) = 0
150 NEXT I
160 LET Q = 0
170 LET W = 0
200 REM  "SEQUENCE CONTROL"
210 IF P = 1 THEN 300
220 IF P = 2 THEN 500
230 LET P = 1
240 GOTO 210
300 REM  "HUMAN'S PLAY"
310 LET P = 2
320 LET Q = 1
330 GOSUB 6000
340 GOSUB 4000
350 IF Q = 0 THEN 600
360 LET Q = 0
370 LET W = 0
380 GOSUB 2200
390 IF W = 3 THEN 700
```

```
400 GOTO 200
500 REM  "COMPUTER'S PLAY"
510 LET P = 1
520 GOSUB 5000
530 GOSUB 3200
540 GOSUB 2400
550 IF W = 30 THEN 800
560 IF Q = 0 THEN 600
570 GOTO 200
600 REM  "TIE GAME"
610 PRINT
620 PRINT "OH WELL"
630 IF P = 2 THEN 660
640 LET P = 2
650 GOTO 900
660 LET P = 1
670 GOTO 900
700 REM  "HUMAN WON"
710 GOSUB 5000
720 PRINT "OOPS - GOT ME"
730 GOTO 900
800 REM  "COMPUTER WON"
810 GOSUB 5000
820 PRINT "HA HA - I WON"
900 REM  "END OF GAME"
910 PRINT "PLAY AGAIN (Y OR N)";
920 INPUT Q$
930 IF Q$ = "Y" THEN 100
940 PRINT
950 PRINT "THANKS FOR PLAYING"
960 PRINT "NAUGHTS AND CROSSES"
970 PRINT "WITH ME.  GOOD-BYE."
980 END
1000 REM  "DESCRIPTION OF GAME"
1010 PRINT "THIS IS TIC-TAC-TOE"
1020 PRINT "YOU ARE FIRST TO PLAY"
1030 PRINT "THEREAFTER WE TAKE TURNS"
1040 PRINT "YOUR SYMBOL IS ALWAYS 'X' AND"
1050 PRINT "MINE IS 'O'"
1060 PRINT "ENTER THE NUMBER OF THE"
1070 PRINT "BOARD POSITION FOR 'YOUR MOVE'"
1080 PRINT "POSITION NUMBERS ARE"
```

```
1090 PRINT "PRINTED ON THE LEFT AND"
1100 PRINT "THE BOARD IS TO THE RIGHT"
1110 PRINT "THREE-IN-A-ROW FOR EITHER"
1120 PRINT "OF US WINS THE GAME."
1130 PRINT "READY";
1140 INPUT Q$
1150 PRINT "THEN HERE WE GO..."
1160 RETURN
2000 REM  "SET UP ROW CHECKER"
2010 DATA 1,2,3,4,5,6,7,8,9
2020 DATA 1,4,7,2,5,8,3,6,9
2030 DATA 3,5,7,1,5,9
2040 FOR I = 1 TO 8
2050 READ R(I,1), R(I,2), R(I,3)
2060 LET R(I,4) = R(I,1)+R(I,2)+R(I,3)
2070 NEXT I
2080 RETURN
2200 REM  "HUMAN INPUT"
2210 PRINT "YOUR MOVE";
2220 INPUT Q
2230 IF Q < 1 THEN 2250
2240 IF Q < 10 THEN 2270
2250 PRINT "ILLEGAL"
2260 GOTO 2220
2270 IF H(Q) <> 0 THEN 2250
2280 IF C(Q) <> 0 THEN 2250
2290 LET H(Q) = Q
2300 LET B(Q) = 1
2400 REM  "CHECK FOR WINNER"
2410 FOR I = 1 TO 8
2420 LET W = B(R(I,1))+B(R(I,2))+B(R(I,3))
2430 IF W = 3 THEN 2460
2440 IF W = 30 THEN 2460
2450 NEXT I
2460 RETURN
3000 REM  "CHECK FOR DEFENSE"
3010 FOR I = 1 TO 8
3020 LET W = B(R(I,1))+B(R(I,2))+B(R(I,3))
3030 IF W = 2 THEN 3060
3040 NEXT I
3050 GOTO 3300
3060 LET Q = R(I,4) - (H(R(I,1))+H(R(I,2))
     +H(R(I,3)))
```

```
3070 LET C(Q) = Q
3080 LET B(Q) = 10
3090 RETURN
3200 REM  "CHECK FOR WIN MOVE"
3210 FOR I = 1 TO 8
3220 LET W = B(R(I,1))+B(R(I,2))+B(R(I,3))
3230 IF W = 20 THEN 3260
3240 NEXT I
3250 GOTO 3000
3260 LET Q = R(I,4) - (C(R(I,1))+C(R(I,2))+C(R(I,3)))
3270 LET C(Q) = Q
3280 LET B(Q) = 10
3290 RETURN
3300 REM  "MAKE OFFENSE PLAY"
3310 IF B(5) <> 0 THEN 3340
3320 LET Q = 5
3330 GOTO 3270
3340 FOR I = 1 TO 9
3350 IF B(I) = 0 THEN 3390
3360 NEXT I
3370 LET Q = 0
3380 RETURN
3390 LET Q = I
3400 GOTO 3270
3500 RETURN
4000 REM  "EMPTY SPACE TEST"
4010 FOR I = 1 TO 9
4020 IF B(I) = 0 THEN 4060
4030 NEXT I
4040 LET Q = 0
4050 RETURN
4060 LET Q = I
4070 RETURN
5000 REM  "COMPUTER'S BOARD"
5010 GOSUB 6000
5020 PRINT "MY MOVE"
5030 RETURN
6000 REM  "DISPLAY BOARD"
6010 PRINT
6020 PRINT "1  2  3";
6030 GOSUB 7000
6040 PRINT "4  5  6";
6050 GOSUB 7020
```

```
6060 PRINT "7  8  9";
6070 GOSUB 7020
6080 RETURN
7000 LET F = 1
7010 LET L = 3
7020 PRINT TAB(10);
7030 FOR I = F TO L
7050 IF C(I) <> I THEN 7070
7060 PRINT "O  ";
7070 IF H(I) <> I THEN 7090
7080 PRINT "X  ";
7090 IF B(I) <> 0 THEN 7110
7100 PRINT ".  ";
7110 NEXT I
7120 PRINT
7130 LET F = L+1
7140 LET L = L+3
7150 RETURN
```

O

There is another game called Reversi, another called Go, and another called Othello. My computer game is dubbed *O-tell-O*. They are checkerboard games.

As the game is played, the markers are constantly changing their colors, belonging first to one player, then the other. In its more prosaic form two-colored tokens are used, usually white on one side and black on the other. When taking an opponent's piece, it is simply turned over, thereby reversing its color. Like most other programmed versions, we use the letters *X* and *O* to distinguish the pieces. Unlike most others, however, this program serves two human combatants. Once you have mastered the strategy with a friend you can easily add a subroutine to take on the computer as an opponent. The design of this program is intended as a base for experiments in artificial intelligence.

THE PLAY

The game begins with four tokens in place as shown in Fig. 0-1. One player will be assigned Xs and the other player Os. The X player always goes first. Each play, including the first one, involves bracketing an opponent's inhabited squares with your own, thereby converting the opponent's pieces to your own. This bracketing can be achieved vertically, horizontally, or diagonally; and any number of squares may be bracketed so long as the sequence of bracketed squares is uninterrupted by blank squares or by squares already owned by the playing individual.

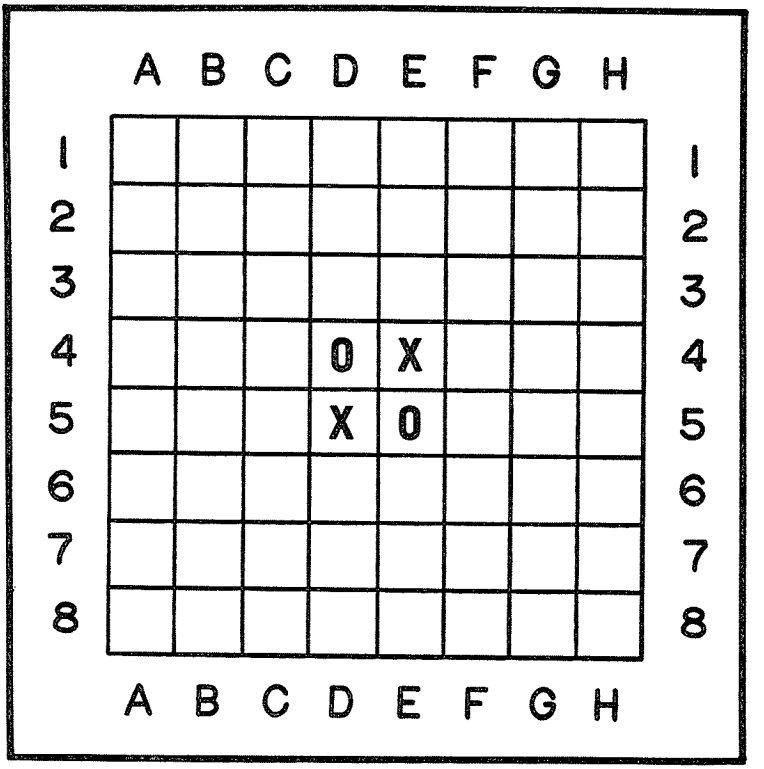


Fig. O-1. The O-tell-O playing board (reduced to eight squares for microcomputer play).

Referring to Fig. 0-1 the first player may bracket the D4 square (containing the opponent's piece) by playing an X at D3, thus converting the Os at D4 to an X and resulting in three vertically aligned Xs at D3, D4, and D5. Or he or she could play an X at C4, thus bracketing the D4 square horizontally (resulting in three horizontally aligned Xs at C4, D4, and E4). Assuming the first player makes this latter move the board would look like the setup pictured in Fig. O-2A.

The opponent (O player) now has but one token on the board, and since his or her move must bracket the first player's pieces, the choice of plays is limited: put O at E3, making a vertical row of Os at E3, E4, and E5; place O at C3, making a diagonal line of Os at C3, D4, and E5; or he can place the O at C5, making a horizontal row of Os at C5, D5, and E5. Let's assume he or she does the latter, which rearranges the pieces to the setup pictured in Fig. O-2B.

Now X can play at C6, converting the O at C5 for a vertical line and the O at D5 for a diagonal line in a single move. Or play X at E6,

similarly converting two lines to Xs. (The first of these is the move pictured in Fig. 0-2C.)

If O plays at B5 he or she brackets the C5 and D5 Xs, converting them both to Os. The play goes on in this fashion, each player depositing one token per move, until the board's squares are filled.

If a move is possible, even if there is only one such possibility you must do it. If you can not make any bracketing move (a move that will result in at least one capture) you must pass. If ever it does happen that neither player can make a legal move the game is over. In the end whoever has the most markers showing is the winner.

PROBLEM DEFINITION

First, there is the matter of a playing board. Internally, perhaps, the board may be mapped as an array (other schemes are possible). It is necessary to have an input scheme that permits a player to specify grid coordinates, such as a column and row number. Several other games in this book, (notably Gunners and Hotshot) use grids, and their conventions expect two numbers to be input, separated by a comma. There is a human factors problem with such conventions, however.

Which comes first, the column number or the row? In fact, which are the columns, and which are the rows? Up-and-down, or across? These are elementary and basic to programmers and to computers, but they are recurring questions in the minds of children and more than just a few adults.

The model of *O-tell-O* provided in this chapter offers a two-pronged solution to the quandary that sometimes occurs at the keyboard. Instead of two numbers the player is asked for a number and a letter. And it doesn't matter which comes first. The computer knows the difference.

Beyond that, and the usual mechanics of managing a two-player game, the programming problem has two major elements. First, there are those implications having to do with rows. Looking at an array the scanning logic must track vertically, horizontally, and on the two diagonals. The other half of the principal programming problem is that attempted plays must be fully qualified, according to the rules, before the array is updated. In short these are the things that must be done following an entry—but mostly before anything in storage is altered.

- Are the column and row coordinates legal? (Not less than one, nor more than eight.)

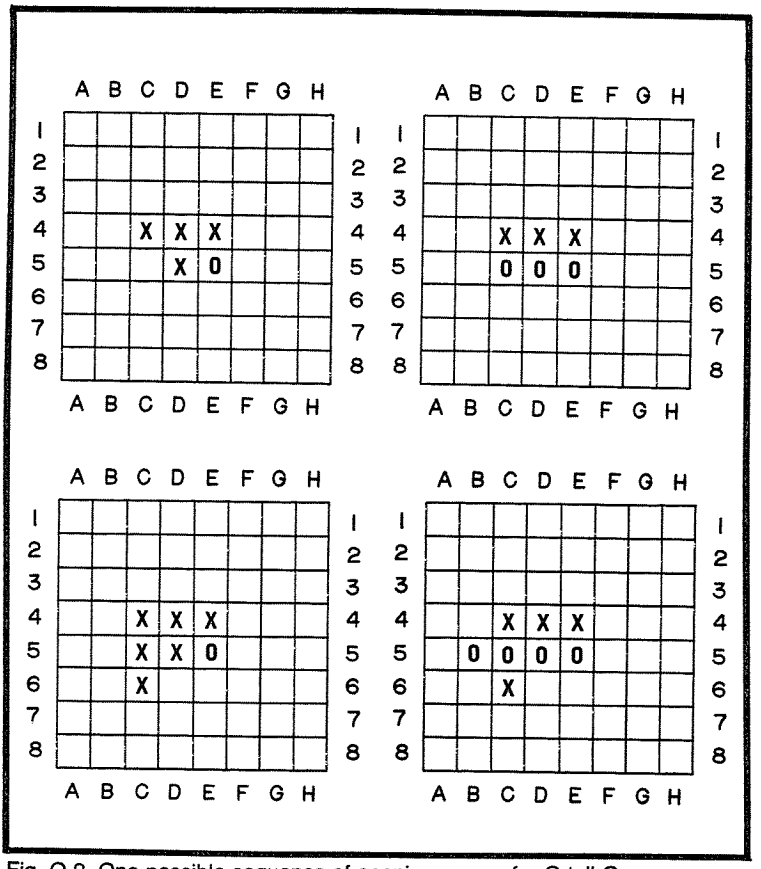


Fig. O-2. One possible sequence of opening moves for O-tell-O.

- Is the indicated position vacant?
- Is that spot adjacent to an opponent's piece&
- If the attempted move is valid thus far, does it bracket? Is there a row eminent from here, in some direction, that ends with this player's piece?
- Is there along that line of tokens one or more of the opposition's uninterrupted pieces that can be reversed?
- If all of the above questions can be answered with yes the move choice can be accepted and applied, but there is more to do.
- Are there any other rows affected by this move? Adroit tacticians can, and will make dual or even triple-thrust moves.

The toughest part of this problem has to do with the need to scan from any point on the board in any of eight possible directions—

every time. The method chosen for doing this, in this program, is looked at next. Everything else about the program's design sort of falls into place based on this relatively simple scheme.

THE BASIC LOGIC PRINCIPLES

The first design decision that was made was to store the board in memory as an array, eight columns wide, and eight rows deep. A conceptual picture of the memory board is shown in Fig. O-3.

The illustration also shows that the axes along which scanning must be done are thought to be like a compass. To be able to do a scan implies the need for looping techniques, and the labels such as east, south, and so on are handy for describing direction.

For example, to do a scan from position 4, 6 toward the east a loop can be used that increments the six (the column or *J* value), maintaining the row-half (four) constant. In the same way a scan toward the west would hold the first number constant and decrement the column coordinate.

For scanning along the north-south axis the same principle applies, but it is the row number that is incremented or decremented, and the column number is held steady. The diagonals pose only a slightly more complex problem: both coordinates must be manipulated.

Looping from the northwest down and to the right, to the southeast, works by incrementing both coordinates in parallel. To back up along the same diagonal, decrementing is used instead of incrementing. To operate on the other diagonal (southwest to northeast) is a little trickier. In that case one coordinate must be incremented as the other is decremented. Directional control along this diagonal is achieved according to which of the two coordinates is added to and which is subtracted from. In either case the arithmetic done on the column and row numbers is always opposite to one another.

The scan process itself is the same from any given point, toward any of the eight possible directions. By starting the loop with the next adjacent position, and by placing the following tests up front, if any test fails immediately, nothing further needs to be done (in that direction). The tests and their order are:

- Is the edge of the board about to be encountered?
- Is the next position vacant?
- If there is a marker there does it belong to this player?

If all of these tests fail the only remaining possibility is that a marker is there and that it belongs to the other player. Depending on the

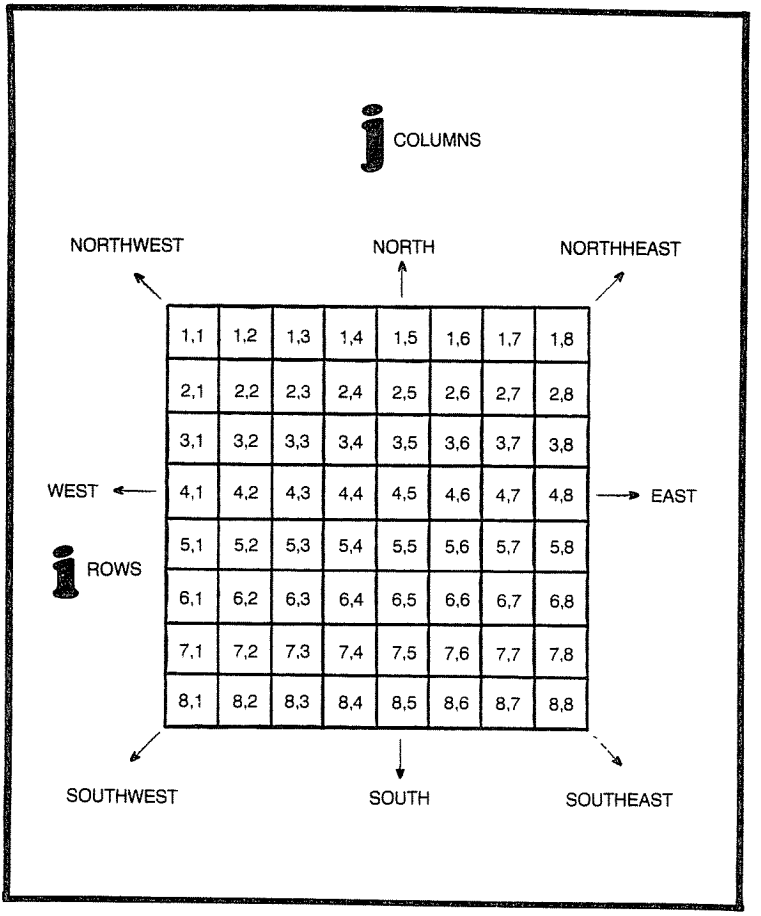


Fig. O-3. The memory board in O-tell-O.

direction that is being worked on the coordinates are adjusted to look at the next position and the tests are tried again. When an exit is finally taken, if the last marker looked at is this player's, a move is appropriate.

At this point the starting and ending points are known. So is the direction. All of the markers along this line can be updated; so another loop is used to overwrite this player's mark in every position.

The total scheme then: scan in each of the eight directions from the indicated position, and update all valid rows. If it does happen that all eight directions are attempted and no capture can be made the choice for this move is not valid. Let the player try again; if necessary, let him or her pass. There is also a simple mechanism

included to detect that rare case when both players have to pass. That ends the game, just the same as when the board is finally full.

Before going on to see how these mechanics work in *O-tell-O* a few points ought to be made about how the program is put together.

A PICTURE OF THE PROGRAM

Most of that to the left of center in Fig. O-4 is like other game programs in this book. There is a hint of something clever in the print board block (3000 to 3170) because of the appurtenance on its right edge. For the moment, however, the right side of the program template deserves attention.

Conceptually the row scanners are categorized into two groups. Those that work on horizontal and vertical axes, and those that work on either of the diagonals. Stair-stepped structures on the right, and their labeling, conveys nearly the whole design of this *O-tell-O* program. Each direction does employ its own string of code, and the overlapping of the blocks with a single connecting thread implies they are all accessed each lap through the player input module. There is also an inference that there must be considerable redundancy of programming statements. There is—but not without a rationale.

The program was originally built with only the east block of coding in place. Once everything worked correctly to that point it took only a short while to copy that routine, modifying it each time only to the extent necessary to change directions. The design philosophy in this case was this: if it works for one, it should work for all.

The advantage of building a program along these lines is that, if there is a bug in the coding that is added later, it must be because of a copying error and not because of a design error. Sometimes the time required for typing in the extra lines of programming is a fair tradeoff for the time that might be required for isolating a logic error in a compact, but extremely complex universal routine.

There is another advantage. This narrative, and the time required to study how the program works are both lessened; all that has to be studied in detail is one of the scanners. From that, you can easily understand how they all work.

O-TELL-O FROM THE INSIDE

Initialization of the program begins with a declaration of the array, B, in statement 30. As shown in the listing, this is an eight-by-eight numeric structure. In the next statement (line 40)

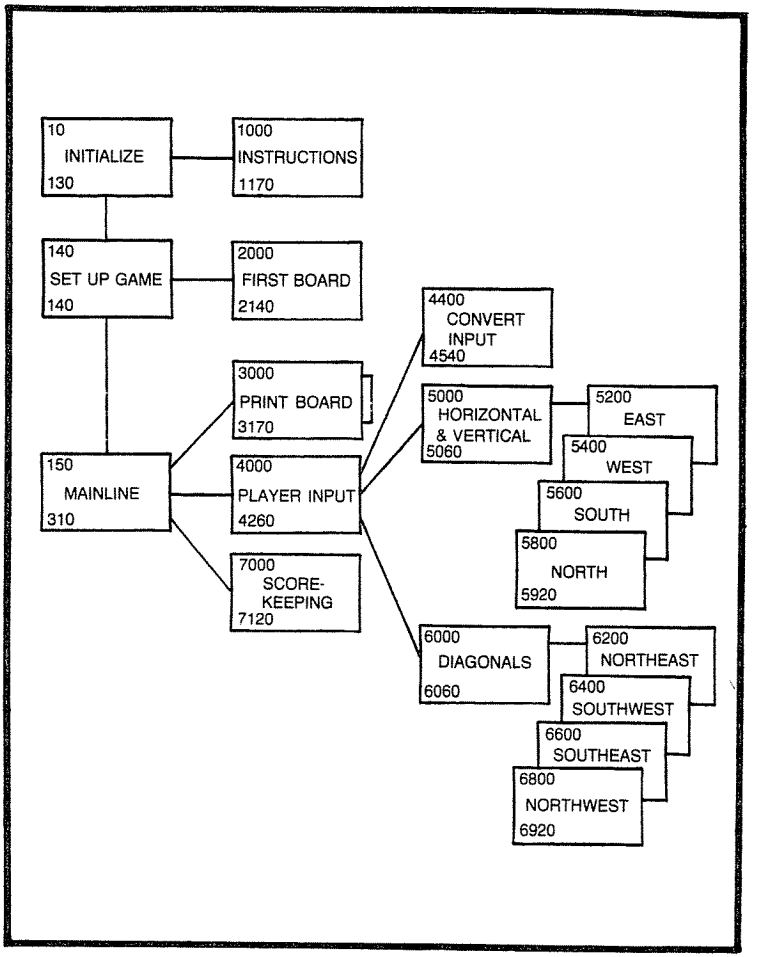


Fig. O-4. Program template for O-tell-O.

there are two alpha tables declared, one called *L* (for letters), and one called *N* (for numbers).

The dual field's READ loop in lines 70, 80, and 90 takes the letters and numbers defined in the DATA statements (lines 50 and 60) and loads them into the *L$* and *N$* tables. Later, as can be surmised, the trick of allowing the player to input a letter and a number coordinate, in either order, works against these two tables to derive pure numeric column and row values.

After the usual bit of dialog (lines 100 through 120) to optionally print the instructions (lines 1000 through 1170) the game is set up by a jump to line 2000. The first board subroutine puts zeros all over

the board (lines 2040 through 2090), establishes the *P* workers, and puts the first four tokens in the center of the board (lines 2100 through 2130).

As to the use of *P*, depending on your thoughts at different points in the program, at times it means *piece*, and other times it means *player*. Most of the time pairs are involved; *P* or *P$* is for one, and *P1* or *P1$* is for the other guy. During the course of play the letters *O* and *X* are alternately exchanged in *P$* and *P1$*, and player numbers 1 and 2 are swapped between *P* and *P1*.

The RETURN in line 2140 goes back to line 150, which is the start of the mainline. The nucleus of the mainline is the repeating sequence of lines 150 through 210. This is basically a dispatch sequence that prints the board, accepts player input, and updates their scores. The first of these tasks is done by the subroutine from line 3000 to 3170.

The principal structure of the print board module is a pair of concentric FOR-NEXT loops, both conditioned to run eight times. The J loop (lines 3040 through 3110) prints a period character for vacant positions or an *O* or an *X*, depending on whether the cells in the *B* array contain a zero, a one, or a two. The *J* loop works across the board, advancing the column number. The outside loop, *I*, works down the board, advancing the row number.

That bit of cleverness promised earlier is revealed by the GOSUB in line 3010. The jump is actually to the tail end of this same subroutine. It is done to make double use of the PRINT statement in line 3140. Just before the board is printed the letters *A* through *H* are output as column labels. After the board is printed, by passing through this same sequence again, the letters are again printed, underneath the board. Coding that jump at the top of the module was easier and cheaper than duplicating the PRINT string.

Notice that row numbers are output also. Line 3030 is responsible for the numbers on the left side of the board, and line 3120 prints the same numbers on the right and advances the printed output by one line.

The only other thing done by this routine is the setting of *T* to zero (line 3160). This is a trigger worker used to know about impending stalemates, where both players find it necessary to pass. It is safe to always reset the trigger to zero whenever the board is printed because the board is not printed whenever either player passes. By simply adding a one to the trigger elsewhere, if it ever reaches two, then both players must have passed in immediate succession.

The next jump from the mainline (line 160) is to line 4000 to accept player input. This module (lines 4000 through 4260) is itself a miniature mainline. All of the dialog having to do with a round of play, by either player, is done within the central structure. Whatever is typed by the player is looked at; and if reasonable, an attempt is made to convert the entry to row and column coordinates. The conversion process depends on a free-standing subroutine (lines 4400 through 4540), and if successful two jumps are used to the array scanning modules.

As the INPUT sequence begins *Q$* and *Q1$* are space-filled. This is done to clear any residue, and to insure an explicit response from the player. It is possible that nothing will be typed following the prompt that is output by line 4020. Line 4040 checks whether anything was entered; if not the logic assumes a pass was intended.

Any value other than a space character in *Q$* will trigger the jump to line 4400. An eight-times FOR-NEXT loop (lines 4400 through 4430) compares *Q$* with the characters in tables *N$* and *L$*, in parallel. If *Q$* matches any of the numbers 1 through 8 the logic is dispatched to line 4460. If the first entry from the keyboard (in *Q$*) is found to match a letter in the series A through H the dispatch is to line 4500 instead.

The remaining possibility is that *Q$* does not match anything in either table. The player must have typed an invalid response: *I* is set to zero as an indicator, and an immediate return is made back to the dialog sequence.

Two other loops are used to get the other coordinate, depending on whether *Q$* was found to have the number or the letter half. Both of these loops (lines 4460 through 4480 and 4510 through 4530) work like the previous one. If *Q1$* is found to be invalid, *I* is zeroed and the indicator is returned. If *Q1$* is found to match an appropriate symbol, the return is executed with a significant value in *I*. The net result is that *I* will have in it the row coordinate and *J* will be returned with the column coordinate.

That test in line 4060 is the one that detects whether *I* came back loaded with something useful or not. If not a branch to line 4000 lets the player go again after he is told: "I DON'T UNDERSTAND".

The next test (lines 4090 through 4110) determines whether the attempted play is aimed at any empty spot on the board. If not the message is terse: "OCCUPIED SPACE". Go again—back to line 4000.

Another variable—*E* for error—is used to keep track of whether all of the row scanners fail or not. Before that process is started, *E* is preset to zero in line 4120. If it comes back as zero no

move was possible—so go again to line 4000. You must have tried an "ILLEGAL MOVE".

There is a short dispatcher that extends from line 5000 to line 5060 to sequence the scanners that work along the horizontal and vertical rows. The entire dispatcher is a FOR-NEXT loop that runs four times—once for each of the directions east, west, south, and north. The variable *K* is used as a loop control (*I* and *J* are otherwise committed). The reason a loop is used is that a pair of temporary workers are needed for the row and column pointers. (*R* and *R1* are for rows; *C* and *C1* are for columns.) All four of these have to be reestablished for each of the directional scanners. Setting them up within a loop does save some repetitious code.

Looking at the east module it can be seen in line 5200 how it is that the dispatcher is able to call successive subroutines with a single GOSUB address (in line 5050: GOSUB 5200). If *K* has a one in it, east will execute. If *K* is not equal to one control is passed to the west; and so on down there also.

To scan along the east-west line, the column number (now in *C*) is the one that should vary. Whatever is in the row worker (now in *R*) will be held constant. Looking specifically toward the east the following logic increments *C*. (Later, to look toward the west, *C* is decremented.)

Line 5210 tests to see whether the attempted play is next to the edge of the board. If so, exit. Line 5220 tests to see whether the next position is vacant. If so, exit.

Line 5230 tests to see whether the next position is occupied by this player's piece. If so, and this is the first time this test was made (meaning *C* has not yet been incremented beyond what it started with), exit by reason of the test in line 5270.

Now increment *C*. This is done in line 5240. The same set of tests are repeated, then, until either an exit is taken or line 5290 is reached. If the flow does get to there a move is in order. At that point *C* has been incremented so that it is larger than *C1*. (*C1* was present to contain the same value as *C*, that value being the *I* column number.)

To make the move, line 5300 overwrites *P*—the player's piece, either a one or a two—into the first row segment location. Line 5310 then increments *C1* and the overwriting continues until *C1* catches up to *C*. When done the exit route that is taken is through line 5330, which increments *E*. When control is finally returned to the dialog area it will be known that a move was made because the error worker will not be zero.

Each of the other scanners work just like this one. Their only differences are in the varying of *R, R1, C,* and *C1*. The logic is always the same; there is even a similarity in the line numbering. If you do encounter any bugs you must have copied something wrong. The design has to be right: "What works for one should work for all."

Back in the mainline two test sequences work to know when the game is over. The one in lines 170 and 210 through 240 takes care of stalemates or voluntary aborts by reason of successive passes. The other test depends on knowing whether the board is all filled up. The scoring routine can detect if it is, and if so, a *W* (for *win*) flag is set to one.

As each play is made a jump to line 7000 is made to update the scores. Concentric loops are used to run across and down the whole board, and as this is being done the player's accumulators are generated so as to have a current count of their pieces. The variable *O* is used for one, and *X* is for the other. If they both add up to 64 *W* is turned on.

That is really all there is to the programming of O-tell-O. These are all of the mechanics for managing the board and for qualifying player's moves. You'll have no trouble developing a ten by ten board from the eight by eight model described if that's your bent.

To develop a version that has the computer as an opponent notice that all of the scanner logic can be used for attempted plays by the computer. The strategies the computer can employ and the logic by which it can deduce its next move is up to you and the degree of artificial intelligence you can impart to your mechanical friend.

Meanwhile, the kids can have fun playing this version while you have fun teaching the computer to play.

THE PROGRAM

```
 10 REM  "OTELLO"
 20 REM
 30 DIM B(8,8)
 40 DIM L$(8), N$(8)
 50 DATA A, 1, B, 2, C, 3, D, 4
 60 DATA E, 5, F, 6, G, 7, H, 8
 70 FOR I = 1 TO 8
 80 READ L$(I), N$(I)
 90 NEXT I
100 PRINT "WANT INSTRUCTIONS (Y OR N)";
110 INPUT Q$
```

```
120 IF Q$ = "N" THEN 140
130 GOSUB 1000
140 GOSUB 2000
150 GOSUB 3000
160 GOSUB 4000
170 IF T <> 0 THEN 210
180 GOSUB 7000
190 IF W = 1 THEN 250
200 GOTO 150
204 IF Q$ = " " THEN 160
206 GOTO 150
210 IF T = 1 THEN 204
220 PRINT "STALEMATE (Y OR N)";
230 INPUT Q$
240 IF Q$ <> "Y" THEN 150
250 PRINT "SCORES   0 = "0"    X = "X
260 PRINT
270 PRINT "ANOTHER GAME (Y OR N)";
280 INPUT Q$
290 IF Q$ = "Y" THEN 140
300 PRINT "BYE"
310 END
1000 PRINT "TWO PLAYERS COMPETE ON AN"
1010 PRINT "8-BY-8 BOARD.  ONE HAS 'X', THE"
1020 PRINT "OTHER HAS 'O'.  IN YOUR TURN,"
1030 PRINT "PLACE YOUR MARK NEXT TO ONE OF"
1040 PRINT "HIS SO THAT YOU MAKE A 'ROW'"
1050 PRINT "(VERTICAL, HORIZONTAL, DIAGONAL)"
1060 PRINT "WITH YOUR'S ON THE ENDS."
1070 PRINT "WHEN YOU DO, YOU 'FLIP' ALL OF"
1080 PRINT "HIS TO BE YOUR'S."
1090 PRINT "IF YOU CAN'T PLAY, PASS (NO"
1100 PRINT "ENTRY)."
1110 PRINT "GAME IS OVER WHEN YOU BOTH HAVE"
1120 PRINT "TO PASS, OR THE BOARD IS FULL."
1130 PRINT "THE PLAYER WITH THE MOST PIECES"
1140 PRINT "WINS THE GAME."
1150 PRINT ".........READY";
1160 INPUT Q$
1170 RETURN
1999 REM  "FIRST BOARD"
2000 LET P$ = "O"
```

```
2010 LET P1$ = "X"
2020 LET P = 1
2030 LET P1 = 2
2040 LET W = 0
2050 FOR I = 1 TO 8
2060 FOR J = 1 TO 8
2070 LET B(I,J) = W
2080 NEXT J
2090 NEXT I
2100 LET B(4,4) = P
2110 LET B(5,5) = P
2120 LET B(4,5) = P1
2130 LET B(5,4) = P1
2140 RETURN
2999 REM  "PRINT BOARD"
3000 PRINT
3010 GOSUB 3140
3020 FOR I = 1 TO 8
3030 PRINT I" ";
3040 FOR J = 1 TO 8
3050 IF B(I,J) <> 0 THEN 3070
3060 PRINT ". ";
3070 IF B(I,J) <> 1 THEN 3090
3080 PRINT "0 ";
3090 IF B(I,J) <> 2 THEN 3110
3100 PRINT "X ";
3110 NEXT J
3120 PRINT I
3130 NEXT I
3135 PRINT
3140 PRINT "    A B C D E F G H"
3150 PRINT
3160 LET T = 0
3170 RETURN
3999 REM  "PLAYER INPUT"
4000 LET Q$ = " "
4010 LET Q1$ = Q$
4020 PRINT P$" MOVE (C,R)";
4030 INPUT Q$, Q1$
4040 IF Q$ = " " THEN 4180
4050 GOSUB 4400
4060 IF I > 0 THEN 4090
4070 PRINT "I DON'T UNDERSTAND"
```

```
4080 GOTO 4000
4090 IF B(I,J) = 0 THEN 4120
4100 PRINT "OCCUPIED SPACE"
4110 GOTO 4000
4120 LET E = 0
4130 GOSUB 5000
4140 GOSUB 6000
4150 IF E <> 0 THEN 4200
4160 PRINT "ILLEGAL MOVE"
4170 GOTO 4000
4180 PRINT "PASS"
4190 LET T = T+1
4200 LET Q = P
4210 LET P = P1
4220 LET P1 = Q
4230 LET Q1$ = P$
4240 LET P$ = P1$
4250 LET P1$ = Q1$
4260 RETURN
4399 REM  "CONVERT INPUT"
4400 FOR I = 1 TO 8
4410 IF Q$ = N$(I) THEN 4460
4420 IF Q$ = L$(I) THEN 4500
4430 NEXT I
4440 LET I = 0
4450 RETURN
4460 FOR J = 1 TO 8
4470 IF Q1$ = L$(J) THEN 4450
4480 NEXT J
4490 GOTO 4440
4500 LET J = I
4510 FOR I = 1 TO 8
4520 IF Q1$ = N$(I) THEN 4450
4530 NEXT I
4540 GOTO 4440
4999 REM  "HORIZONTAL & VERTICAL"
5000 FOR K = 1 TO 4
5010 LET R = I
5020 LET R1 = I
5030 LET C = J
5040 LET C1 = J
5050 GOSUB 5200
5055 NEXT K
```

```
5060 RETURN
5199 REM  "EAST"
5200 IF K <> 1 THEN 5400
5210 IF C+1 > 8 THEN 5340
5220 IF B(R,C+1) = 0 THEN 5340
5230 IF B(R,C+1) = P THEN 5270
5240 LET C = C+1
5260 GOTO 5210
5270 IF C = J THEN 5340
5290 IF C1 > C THEN 5330
5300 LET B(R,C1) = P
5310 LET C1 = C1+1
5320 GOTO 5290
5330 LET E = E+1
5340 RETURN
5399 REM  "WEST"
5400 IF K <> 2 THEN 5600
5410 IF C-1 < 1 THEN 5340
5420 IF B(R,C-1) = 0 THEN 5340
5430 IF B(R,C-1) = P THEN 5470
5440 LET C = C-1
5460 GOTO 5410
5470 IF C = J THEN 5340
5490 IF C1 < C THEN 5330
5500 LET B(R,C1) = P
5510 LET C1 = C1-1
5520 GOTO 5490
5599 REM  "SOUTH"
5600 IF K <> 3 THEN 5800
5610 IF R+1 > 8 THEN 5340
5620 IF B(R+1,C) = 0 THEN 5340
5630 IF B(R+1,C) = P THEN 5670
5640 LET R = R+1
5660 GOTO 5610
5670 IF R = I THEN 5340
5690 IF R1 > R THEN 5330
5700 LET B(R1,C) = P
5710 LET R1 = R1+1
5720 GOTO 5690
5799 REM  "NORTH"
5800 REM
5810 IF R-1 < 1 THEN 5340
5820 IF B(R-1,C) = 0 THEN 5340
```

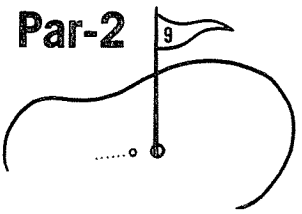
```
5830 IF B(R-1,C) = P THEN 5870
5840 LET R = R-1
5860 GOTO 5810
5870 IF R = I THEN 5340
5890 IF R1 < R THEN 5330
5900 LET B(R1,C) = P
5910 LET R1 = R1-1
5920 GOTO 5890
5999 REM  "DIAGONALS"
6000 FOR K = 1 TO 4
6010 LET R = I
6020 LET R1 = I
6030 LET C = J
6040 LET C1 = J
6050 GOSUB 6200
6055 NEXT K
6060 RETURN
6199 REM  "NORTHEAST"
6200 IF K <> 1 THEN 6400
6210 IF C+1 > 8 THEN 6340
6215 IF R-1 < 1 THEN 6340
6220 IF B(R-1,C+1) = 0 THEN 6340
6230 IF B(R-1,C+1) = P THEN 6270
6240 LET C = C+1
6245 LET R = R-1
6260 GOTO 6210
6270 IF R = I THEN 6340
6290 IF C1 > C THEN 6330
6300 LET B(R1,C1) = P
6310 LET C1 = C1+1
6315 LET R1 = R1-1
6320 GOTO 6290
6330 LET E = E+1
6340 RETURN
6399 REM  "SOUTHWEST"
6400 IF K <> 2 THEN 6600
6410 IF C-1 < 1 THEN 6340

6415 IF R+1 > 8 THEN 6340
6420 IF B(R+1,C-1) = 0 THEN 6340
6430 IF B(R+1,C-1) = P THEN 6470
6440 LET C = C-1
6445 LET R = R+1
```

```
6460 GOTO 6410
6470 IF R = I THEN 6340
6490 IF C1 < C THEN 6330
6500 LET B(R1,C1) = P
6510 LET C1 = C1-1
6515 LET R1 = R1+1
6520 GOTO 6490
6599 REM  "SOUTHEAST"
6600 IF K <> 3 THEN 6800
6610 IF R+1 > 8 THEN 6340
6615 IF C+1 > 8 THEN 6340
6620 IF B(R+1,C+1) = 0 THEN 6340
6630 IF B(R+1,C+1) = P THEN 6670
6640 LET C = C+1
6645 LET R = R+1
6660 GOTO 6610
6670 IF R = I THEN 6340
6690 IF C1 > C THEN 6330
6700 LET B(R1,C1) = P
6710 LET C1 = C1+1
6715 LET R1 = R1+1
6720 GOTO 6690
6799 REM  "NORTHWEST"
6800 REM
6810 IF R-1 < 1 THEN 6340
6815 IF C-1 < 1 THEN 6340
6820 IF B(R-1,C-1) = 0 THEN 6340
6830 IF B(R-1,C-1) = P THEN 6870
6840 LET C = C-1
6845 LET R = R-1
6860 GOTO 6810
6870 IF R = I THEN 6340
6890 IF C1 < C THEN 6330
6900 LET B(R1,C1) = P
6910 LET C1 = C1-1
6915 LET R1 = R1-1
6920 GOTO 6890
6999 REM  "SCOREKEEPING"
7000 LET X = 0
7010 LET O = 0
7020 FOR I = 1 TO 8
7030 FOR J = 1 TO 8
7040 IF B(I,J) <> 1 THEN 7060
```

```
7050 LET O = O+1
7060 IF B(I,J) <> 2 THEN 7080
7070 LET X = X+1
7080 NEXT J
7090 NEXT I
7100 IF X+O <> 64 THEN 7120
7110 LET W = 1
7120 RETURN
```

P



This is another of those few games that have evolved as a direct consequence of modern computer technology. This one specifically is attributable to the invention of the serial printer. In other languages, on a variety of systems, this game has also been known as golf, minigolf, putt-putt, and perhaps others but the basis of each is always the same.

The program prints an asterisk (the ball) and a zero (the hole) some distance apart. You simulate putting by guessing the number of print spaces between the ball and the hole. The program then moves the ball, and counts your strokes. With any luck at all (or if you have calibrated eyeballs) you should hit the hole in two strokes. Thus, *par* is *2*.

As programmed here, *Par-2* optionally permits two, three, or four players, and the course is a full eighteen holes. Although it has been done often on Teletype and similar printers this model was developed on a CRT, which permits but thirty-two characters per line. Which does prove that no matter what type of display you have, nor how austere your computing budget may be, you too may adopt this elementary but fun programmers' favorite.

DESIGN CONSIDERATIONS

Because the *Par-2* game is based on printed output there is a certain degree of sensitivity in its design regarding the mechanical attributes of the display used. Line length is one such attribute. It is desired that the ball and the hole always be printed on the same

output line. To achieve this the sum of the player's guess and the TAB value of the ball's current position must be restricted. The range permitted in this model is zero through thirty-one. (Thirty-two character spaces).

There is another system-specific sensitivity to the designing of this program, and it has to do with how PRINT with TAB is implemented. Our BASIC doesn't mind what presentation sequence is expressed in a print statement, but if a first TAB variable is greater than the next the output will be printed as if the second TAB didn't exist in the expression. The significance in *Par-2* is that the ball may have to be printed either to either the left or the right of the hole (the player may overshoot).

The net effect of the TAB convention is best shown with an example. Assume that *B* means ball and *H* means hole. Now consider:

```
PRINT TAB (B) "*" TAB (H) "O"
```

As long as the value in *B* is less than the value in *H* the result will be just as we expect. On the other hand, if *B* had fifteen and *H* contained twelve, the output would look like this:

```
*O
```

Meaning that the asterisk would be printed in position 15; but since the *H* value had already been passed (scanning from left to right) the second TAB is ignored and the hole is dumped immediately.

Persevering programmers shall overcome. The solution used in this example depends on the use of two different print statements. For a given line only one is used. Their selection is argued by conditionally comparing *B* and *H*. If *B* is less than *H*, then:

```
PRINT TAB (B) "*" TAB (H) "O"
```

Whenever the duffer on the keyboard overshoots, meaning *B* will be bumped beyond *H*, then:

```
PRINT TAB (H) "O" TAB (B) "*"
```

There is, of course, a third condition possible. *B* and *H* may be equal. A Palmer-type player may get lucky and ace it, or—even should it take all afternoon—at some point the ball may roll in, causing *B* and *H* to come up alike.

My microcomputer's BASIC will not back up along a print line. Nor can we preclude a one-character escapement after each charac-

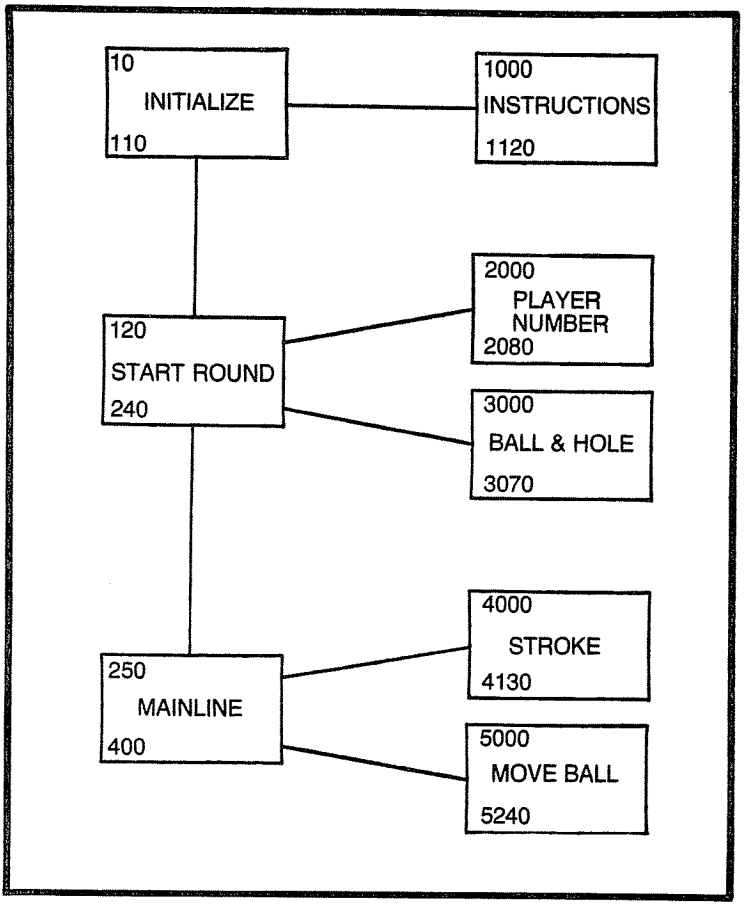


Fig. P-1. Program template for Par-2.

ter. So the ball can never be superimposed over the hole. To show the concluding stroke in a player's turn this program outputs an *X* when the print positioning values become equal. Problem solved.

To an extent at least all of that above had some bearing on the way the tasks were allocated and in the make up of the template in Fig. P-1. There are basically two tasks required to begin a player's turn, and there are two others for repeated strokes within a turn. The illustration can only portray conceptual structure, however.

PROGRAMMING PAR-2

The dimensioning of *P* for five in line 40 portends a bit of subtlety. Up to four players may be accommodated; their scores are maintained in *table P* in the first four slots. The fifth element of *P* is

used to hold the round-of-play counter. There is nothing exciting about this, but it did have to be explained.

The expression in line 120 is now obvious—the game is initialized to start out on hole number 1. The club member is then asked in line 130 to identify how many players are in his party. Only two, three, or four is permitted as a response, and the answer is stored in A. The expression in line 150 is used to clean up whatever is typed into Q, insuring that A won't be a digit followed by a decimal value. (Half a player would be difficult to accommodate.)

The final act in preparing the course is done by the FOR-NEXT loop in lines 190 through 210. Enough of *table P* is erased (A fields) to give each player his or her own scorecard. We are now ready to tee off.

In line 2000, following the GOSUB out of line 230, P will be less than A, so the branch to line 2030 is taken to increment the player number. The simple variable P (as distinguished from *table P* fields) starts out as zero; in the first round it is made a one. In succeeding rounds, when A players have played, P will be reset to zero in line 2010, and *P(5)* will be incremented at that time to show that the party has advanced to the next hole.

Simulating what happens in real life the subroutine starting at line 3000 uses the RND function to set up each player's putting green situation. The ball is always initialized at one, but the length of the putt is controlled by whatever is generated in H. Line 3010 will insure that H is at least a four; line 3020 insures that H is not beyond the end of the print line. D means distance (line 3030), the ball and the hole are displayed (lines 3040 and 3050), and the stroke counter is initialized in S. Back to the mainline for the player's stroke.

Line 250 calls line 4000, and the single-character prompt is #, followed by the system's own question mark, indicating a number is being asked for. Whatever is typed into Q is cleaned up in line 4020; only a positive whole number can be tolerated by the logic that follows.

Either Q is added to B or it is subtracted for comparison purposes in line 4040 or line 4110. This logic permits the player to indicate distance in an absolute sense—the computer is smart enough to know which direction is toward the hole. If the resulting arithmetic won't place the ball off the left or right end of the print line the entry is accepted and the subroutine is exited. If the player's entry is not tolerable he or she is admonished to try again; the stroke does count, but the ball is not moved.

The remaining jump out of the mainline goes to line 5000 to reposition the ball based on how hard it was hit. The first decision is whether the ball is in the hole. If so an *X* is printed, and ACE, PAR, BOGEY, or DUFFER is printed, depending on the number of strokes taken. Notice that in line 5050, six is considered enough. A forced exit is manufactured by making *B* and *H* equal in line 5160 so that the common exit path through line 5050 will let the next player go ahead back in the mainline.

Looking at line 270 this is where distance is important. When there is no distance between the ball and the hole, meaning *D* is zero, the regular loop back to line 250 is quit. In line 280, then, the player's strokes are added from *S* to his or her scorecard. If the last player hasn't yet played line 290 will branch the program back to line 230. After the last player has been up, each time, line 300 checks what hole is being played; if an eighteen is not yet in *P(5)* the game goes on, again back to line 230.

Chances are the option to play again after the final scorecard is printed (by the FOR-NEXT loop in lines 320 through 340) will often be taken. Playing golf on a computer is even less strenuous than riding a real course in one of those motor-driven caddy carts. And like in the real game, on a pleasant afternoon Par-2 can be enjoyed when you are a member of a companionable foursome.

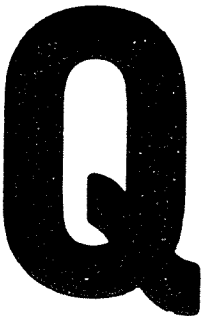
THE PROGRAM

```
 10 REM  "PAR-2"
 20 REM
 30 GOSUB 9000
 40 DIM P(5)
 50 PRINT "WANT THE RULES (Y OR N)";
 60 INPUT Q$
 70 IF Q$ = "Y" THEN 110
 80 IF Q$ = "N" THEN 120
 90 PRINT "HUH?"
100 GOTO 50
110 GOSUB 1000
120 LET P(5) = 1
130 PRINT "HOW MANY PLAYERS (2-4)";
140 INPUT Q
150 LET A = INT(Q)
160 IF A < 2 THEN 180
170 IF A < 5 THEN 200
180 PRINT "HUH?"
190 GOTO 130
```

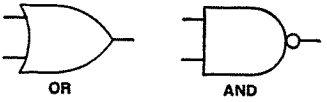
```
200 FOR I = 1 TO A
210 LET P(I) = 0
220 NEXT I
230 GOSUB 2000
240 GOSUB 3000
250 GOSUB 4000
260 GOSUB 5000
270 IF D <> 0 THEN 250
280 LET P(P) = P(P)+S
290 IF P <> A THEN 230
300 IF P(5) <> 18 THEN 230
310 PRINT "SCORE CARD"
320 FOR I = 1 TO A
330 PRINT "#"I"="P(I)
340 NEXT I
350 PRINT
360 PRINT "ANOTHER ROUND (Y OR N)";
370 INPUT Q$
380 IF Q$ = "Y" THEN 120
390 PRINT "SO LONG GANG."
400 END
1000 PRINT "PLAYERS TAKE TURNS.  I WILL"
1010 PRINT "PRINT A BALL AND A HOLE.  GUESS"
1020 PRINT "THE DISTANCE (PRINT SPACES) FROM"
1030 PRINT "THE BALL TO THE HOLE."
1040 PRINT
1050 PRINT "YOUR TURN CONTINUES UNTIL YOU"
1060 PRINT "SINK THE BALL, OR UNTIL YOU HAVE"
1070 PRINT "SHOT SIX TIMES."
1080 PRINT
1090 PRINT "A COMPLETE GAME IS 18 HOLES."
1100 PRINT "LOWEST SCORE WINS."
1110 PRINT
1120 RETURN
1999 REM  "PLAYER NUMBER"
2000 IF P < A THEN 2030
2010 LET P = 0
2020 LET P(5) = P(5)+1
2030 LET P = P+1
2040 PRINT "HOLE"P(5);
2050 PRINT "  PLAYER #"P;
2060 PRINT "  SCORE ="P(P)
```

```
2070 PRINT
2080 RETURN
2999 REM  "BALL & HOLE"
3000 LET B = 1
3010 LET H = INT(100*RND(1))+4
3020 IF H > 31 THEN 3010
3030 LET D = H-B
3040 PRINT TAB(B) "*";
3050 PRINT TAB(H) "O"
3060 LET S = 0
3070 RETURN
3999 REM  "STROKE"
4000 PRINT "#";
4010 INPUT Q
4020 LET Q = INT(ABS(Q))
4030 IF B > H THEN 4110
4040 IF B+Q < 32 THEN 4080
4050 PRINT "OFF THE COURSE -- TRY AGAIN."
4060 LET S = S+1
4070 GOTO 4000
4080 LET B = B+Q
4090 LET S = S+1
4100 RETURN
4110 IF B-Q < 0 THEN 4050
4120 LET Q = 0-Q
4130 GOTO 4080
4999 REM  "MOVE THE BALL"
5000 IF B <> H THEN 5150
5010 PRINT TAB(H) "X"
5020 LET M$ = "ACE"
5030 IF S > 1 THEN 5070
5040 PRINT M$
5050 LET D = H-B
5060 RETURN
5070 LET M$ = "PAR"
5080 IF S > 2 THEN 5100
5090 GOTO 5040
5100 LET M$ = "BOGEY"
5110 IF S > 3 THEN 5130
5120 GOTO 5040
5130 LET M$ = "DUFFER"
5140 GOTO 5040
5150 IF S < 6 THEN 5180
```

```
5160 LET H = B
5170 GOTO 5130
5180 IF B > H THEN 5220
5190 PRINT TAB(B) "*";
5200 PRINT TAB(H) "0"
5210 GOTO 5050
5220 PRINT TAB(H) "0";
5230 PRINT TAB(B) "*"
5240 GOTO 5050
9000 REM  "RANDOM NUMBER ROUTINE"
9010 LET Z = RND(1)
9020 RETURN
```



Quantal



Quantal; An adjective meaning data that fits into only one of two categories. Typical quantal elements are yes or no, true or false, on or off, all or none.

This game has two *quantal* properties. Each player has ten switches; some may be on and some may be off. The object of the game is to get all of your switches set alike at the same time—either all on or all off. If you do so before your opponent does you win the game.

Problem: the switches are all hidden. In the beginning the program sets up all of both player's switches—purely at random. Some may be turned on and some may be turned off. It is even possible (in theory at least) that they could all be initialized to the same setting.

In your turn you may interrogate only one switch. Any one. This is done by entering a number between one and ten, inclusive. The response will be simply *ON* or *OFF*, as the case may be for that switch. You are then afforded the opportunity to leave the switch as you found it or ask that it can be changed to its opposite setting.

The program doesn't look at all of a player's switches until after he or she has taken a turn. Because all ten could be on (or off) at the outset, for the first turn at least, any one switch should be checked, but left alone. There doesn't seem to be any good reason for changing the state of that first switch. It could be correct as is.

A player could win right from the start, but he probably won't. The randomizing scheme that is used sets up the switches individu-

ally. The odds that ten random settings (in a row) will come up the same must be astronomical. But it could happen.

After the first turn both players are entirely on their own. One might adopt a strategy of setting all switches to the same position as that of the first one looked at. This would produce a winner in a maximum of ten rounds. Maybe even less. Only nine turns would be required if the final switch is already in the correct state. In the same way only eight turns would be required if the two switches left for last had been initialized to match the chosen pattern. Equally maybe, however, Lady Luck could favor the opposition.

Simple strategies and a lot of luck. But the kids will love it—and you too, for the short hour that you will spend adding this program to your library.

DESIGN & LOGIC

As shown by the program's template in Fig. Q-1 the design of *Quantal* is nearly artless. Like most of the others in this book the program is shown to provide instructions out of the initialization sequence. This is followed by a brief start game sequence which includes jumps to two modules that do housekeeping tasks. The real mainline follows, and it consists mainly of a cyclic calling sequence to the two primary supporting subroutines. That is really all there is to the layout of *Quantal*.

Looking now at the program's listing there is a clue in line 40 that there is a slight innovation internally. A table is used to hold the sets of switches. But notice that only one table is defined (*table S*), and it has twenty elements. Half of these are used for one player and the remaining ten elements belong to the second player.

One benefit of this design approach can be seen in the short subroutine that does the randomized settings. A single FOR-NEXT loop (lines 2000 through 2050) runs twenty times, putting either a one or a zero in each spot, with no distinction as to which player will ultimately have the upper or lower half of the table. In fact it is the tail end of this module that sets up *P* (for player) to condition which player will have the first turn.

Notice also the optimum use of the coding shown here. When all twenty switches are set the loop completes and falls through. There is some residue left in the *R* variable at that point, and the conditional in line 2070 tests it, causing *P* to end up with either a one or an eleven in it. These two codes are interpreted elsewhere to control which player is up, but more importantly the values of one and eleven enable *P* to be used as a base subscript for accessing the shared table.

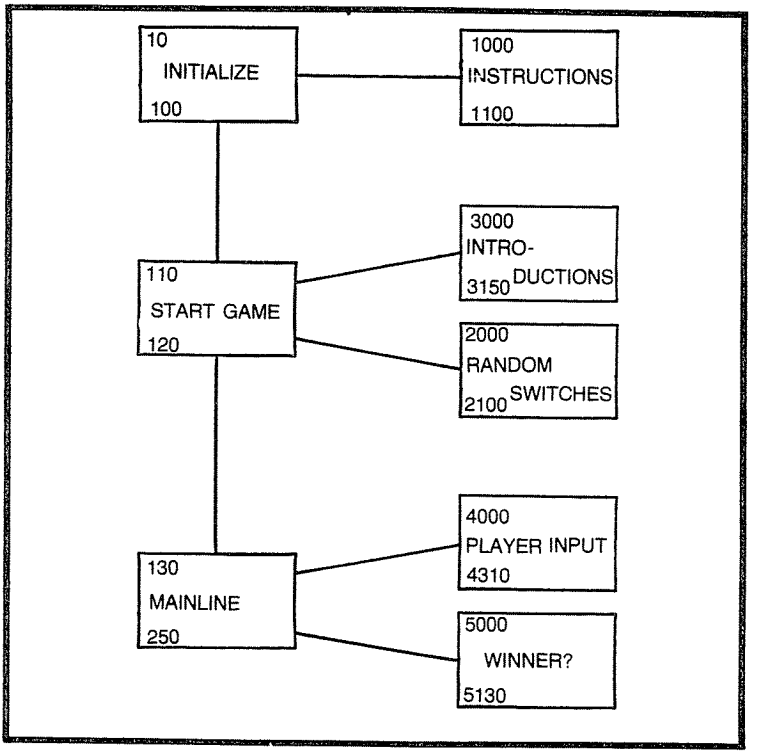


Fig. Q-1. Program template for Quantal.

At least some of the savings gained by this cleverness is traded off in the rote dialog that extends from line 3000 to line 3150. Most players seem to enjoy the personal touch afforded by having the computer "know" them, however. This subroutine asks them their names and insists that at least the first two letters of each are in fact letters, and that the two names are not alike. Later, each player is called to the keyboard for his or her turn, by whatever nickname was given during the introductions sequence.

Internally the players are still just numbers. Nearly the whole game is accomplished by the player input module, which begins in line 4000, and the initial logic there determines which player is up.

The prompt that follows the player's name is: "SWITCH #". The *Q* variable will receive whatever is typed, and from line 4060 through 4140 the entry is checked thoroughly to insure that it is logically acceptable.

Once execution gets as far as line 4150 the entry has to be one of the numbers from one to ten, inclusive, so any switch can be accessed as requested. A single subscript (*S*) is formed by adding

the *P* code (one or eleven) to the keyboard entered number, less one. The conditional in line 4160 then examines the table (one through ten, *or* eleven through twenty) and causes "ON" or "OFF" to be printed.

The second step of each player's turn begins with the simple prompt: "CHANGE", or "CHANGE (Y OR N)". In line 4210 the *C$* field may either be blank or it may contain the self-erasing yes-or-no note. This extra feature is included to suppress the monotonous reminder of what response is expected.

If the player does answer with an N an immediate exit is taken out of line 4240, down to the RETURN in line 4290. On the other hand, if the player wants to change the switch being looked at, the sequence from line 4270 to 4310 will flip the switch that is referenced still, by subscript *S*. In both cases the RETURN will be back to the mainline area.

The last call out of the mainline is to line 5000. A pair of loops is used in this module to quickly scan both players' table halves to see if either has won. Both processes work alike. The *Q* variable is used as a temporary worker to hold the topmost switch of a player's group. Each succeeding switch is then compared with the first, and the loop is broken if any mismatch is found. When finally either loop does complete, meaning that all ten elements are the same (either zeros or ones), the winner is announced by name. The exit sequence is common to either loop, through line 5060, which loads *Q* with an arbitrary code of *99*. It is this code that is looked for in line 130 in the mainline to stop the game.

The balance of the mainline includes two options. The program can be ended or another game started. If the continue option is called for a second sequence allows conditionally branching to call for new player's names, or the same two may compete again.

Chances are they will want to play again. This is a simple game, but the "sudden win" aspect keeps them coming back.

THE PROGRAM

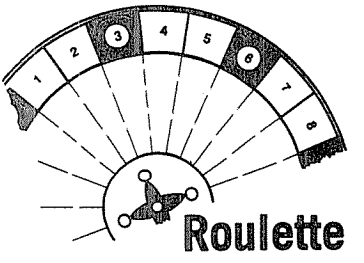
```
10 REM  "QUANTAL"
20 REM
30 GOSUB 9000
40 DIM S(20)
50 PRINT "WANT INSTRUCTIONS (Y OR N)";
60 INPUT Q$
70 IF Q$ = "Y" THEN 100
80 IF Q$ = "N" THEN 110
90 GOTO 50
```

```
 100 GOSUB 1000
 110 GOSUB 3000
 120 GOSUB 2000
 130 IF Q = 99 THEN 170
 140 GOSUB 4000
 150 GOSUB 5000
 160 GOTO 130
 170 PRINT
 180 PRINT "ANOTHER GAME (Y OR N)";
 190 INPUT Q$
 200 IF Q$ = "N" THEN 240
 210 PRINT "SAME PLAYERS (Y OR N)";
 215 INPUT Q$
 220 IF Q$ = "N" THEN 110
 230 GOTO 120
 240 PRINT "BYE"
 250 END
1000 PRINT "TWO PLAYERS EACH HAVE 10"
1010 PRINT "SWITCHES.  SOME ARE ON AND"
1020 PRINT "SOME ARE OFF.  IN A TURN YOU"
1030 PRINT "MAY TEST ONE OF YOUR SWITCHES"
1040 PRINT "AND LEAVE IT AS IS, OR YOU"
1050 PRINT "MAY CHANGE IT.
1060 PRINT
1070 PRINT "THE FIRST PLAYER TO HAVE HIS 10"
1080 PRINT "ALL ON, OR ALL OFF, WINS."
1090 PRINT
1100 RETURN
1999 REM  "20 RANDOM SWITCHES"
2000 FOR I = 1 TO 20
2010 LET R = INT(10*RND(1))
2020 LET S(I) = 0
2030 IF R < 5 THEN 2050
2040 LET S(I) = 1
2050 NEXT I
2060 LET P = 1
2070 IF R < 5 THEN 2090
2080 LET P = 11
2090 LET C$ = "(Y OR N)"
2100 RETURN
2999 REM  "INTRODUCTIONS"
3000 PRINT "PLEASE TELL ME YOUR NAMES"
3010 PRINT "PLAYER #1";
```

```
3020 INPUT P1$
3030 PRINT "PLAYER #2";
3040 INPUT P2$
3050 IF P1$ <> P2$ THEN 3080
3060 PRINT "I DON'T UNDERSTAND."
3070 GOTO 3000
3080 IF P1$ < "AA" THEN 3110
3090 IF P2$ < "AA" THEN 3140
3100 RETURN
3110 PRINT "ENTER #1 AGAIN";
3120 INPUT P1$
3130 GOTO 3050
3140 PRINT "ENTER #2 AGAIN";
3150 GOTO 3040
3999 REM  "PLAYER INPUT"
4000 IF P = 1 THEN 4030
4010 LET P = 1
4015 PRINT P2$;
4020 GOTO 4040
4030 LET P = 11
4035 PRINT P1$;
4040 PRINT ":  SWITCH #";
4050 INPUT Q
4060 IF Q > 0 THEN 4090
4070 PRINT "NO MINUS!"
4080 GOTO 4040
4090 IF Q <= 10 THEN 4120
4100 PRINT "TOO LARGE."
4110 GOTO 4040
4120 IF Q = INT(Q) THEN 4150
4130 PRINT "NO DECIMAL!"
4140 GOTO 4040
4150 LET S = P+Q-1
4160 IF S(S) = 1 THEN 4190
4170 PRINT "OFF"
4180 GOTO 4200
4190 PRINT "ON"
4200 PRINT "CHANGE ";
4210 PRINT C$
4212 LET C$ = " "
4214 LET Q$ = C$
4220 INPUT Q$
```

```
4230 IF Q$ = "Y" THEN 4270
4240 IF Q$ = "N" THEN 4290
4250 LET C$ = " (Y OR N)"
4260 GOTO 4200
4270 IF S(S) = 0 THEN 4300
4280 LET S(S) = 0
4290 RETURN
4300 LET S(S) = 1
4310 RETURN
4999 REM  "WINNER?"
5000 LET Q = S(1)
5010 FOR I = 2 TO 10
5020 IF S(I) <> Q THEN 5080
5030 NEXT I
5040 PRINT P2$;
5050 PRINT " JUST WON"
5060 LET Q = 99
5070 RETURN
5080 LET Q = S(11)
5090 FOR I = 12 TO 20
5100 IF S(I) <> Q THEN 5070
5110 NEXT I
5120 PRINT P1$;
5130 GOTO 5050
9000 REM  "RANDOM NUMBER ROUTINE"
9010 LET Z = RND(1)
9020 RETURN
```

R



Outside it is dark, the sun having gone down now some two hours. On deck all that is heard is the frogs on the bank, and the soft lapping of the paddle wheel on the oily bl ck waters of the Mississippi. The year is 1850 and we are aboard a River Queen, just out of New Orleans.

Inside it is warm, and the air hangs heavy with cigar smoke. There are muted sounds of chatter and occasional laughter. Then, above it all, "Ladies and Gentlemen, place your bets please!"

Roulette is the most glamorous of all gambling games. As you watch the whirling ivory ball spin around the wheel, you can almost see gentlemen in tall silk hats and ladies in satin gowns. And perhaps even Lady Luck herself hovers nearby.

In this, our microcomputer version of *Roulette*, certain changes were made to the traditional game. Some were necessitated by computer mechanics; some others were merely my choices. Likely as not, Mark Twain, that champion of the riverboats of old would approve. Hopefully, you will too.

PLAYING ROULETTE

You can bet on any number—zero through thirty-six—and if the number that comes up is the same that you entered you win five times your bet. You may also bet on the color, red or black. There are twelve red numbers and twenty-five black ones; the computer knows which are red because they are the ones that can be evenly

divided by three. If your choice of color is the right one your bet is tripled. If not you lose twice what you bet.

Two other betting options include odd/evens and high/low. If you do guess correctly on whether the number that comes up is odd or even, you win double; otherwise you lose your bet. The high range of numbers is all of those above eighteen, and low is considered those from zero through eighteen. A win or a loss in this case is even money, based on your bet.

After you have placed your markers on each of the four options—you can pass on any of them—you are asked to input your bet. The computer-cum-croupier then spins the wheel. Up comes the results, your winnings (or losses) are tallied, and the game goes on.

To retire and leave town with your winnings, enter a zero bet. The program will ask you if naught was intended; if your answer is affirmative, the game will end. Because there is a variety of options posed, such as this, there is a rather intelligent error-checking feature in the program. A lot of the fun in playing *Roulette*, as provided here, is in seeing the variety of messages that will be printed following careless keystroking.

Computer gamesmen will take notice that this is solitaire, and it ends only on command (the zero bet), with no replay option. These are departures from normal, and there are a few others inside of the program itself. In the study that follows some of its concepts are useful for other game programs.

PROGRAMMING ROULETTE

This program is lengthy but not particularly complex. Its design serves as an example of making tradeoffs of conciseness in favor of simplicity. Much of its length is caused by the amount of dialog needed, but there is also quite a lot of redundancy in its procedures. The program's structure is shown by the template in Fig. R-1. Because the drawing is uncluttered it will serve nicely as a tour map through the labyrinth of coding that makes up Roulette.

Initialize (10—100)

First of significance is line 35. The name *M* is symbolic for money. In this case there are five workers defined. Each of the first four are for options calculations, and the fifth spot in *table M* is for totaling the others into. The only other task that is accomplished by this block of coding is permitting an optional printout of the game's brief description. That's next.

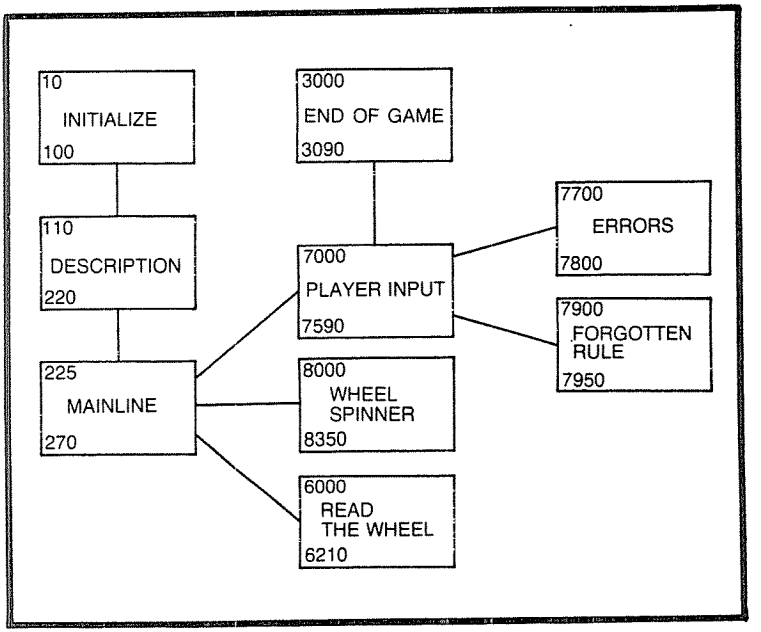


Fig. R-1. Program template for Roulette.

Description (110-220)

There is nothing exotic here—just thirteen print statements. They are either executed or not, depending on the two conditionals back up in lines 60 and 70.

Mainline (225-270)

Line 225 was inserted for debugging reasons. Here the *M* (for money) accumulator is zeroed so that if, while on a bug-chasing expedition, a vectored start is done, *M* will be reset (it should begin with zero automatically whenever the program is first loaded). The rest of the mainline is simply three subroutine jumps, with a loop back through them caused by the GOTO 230 that is in line 270. The calling sequence of lines is 7000, 8000, then 6000. (That is also the order shown in the template.)

Player Input (7000—7590)

After the prompt-line PRINT: "number (0—36) "—the player is supposed to pick a number, if that is his or her choice. The variable *A* is preloaded with 99, just in case a number is not typed in. From line 7020 through 7060 whatever is typed is qualified; and if nothing was the error-checking is bypassed. (This works because of the

ninety-nine test case necessary because zero is a valid player's choice.)

Line 7060 asks for a one-letter code—*B* for black, or *R* for red — and the input will be accepted into *B$*. If the player wishes not to bet on a color, and enters nothing the preloaded space code in *B$* (done by line 7065) will let the program bypass the color-coding logic.

Forgotten Rule

That business of testing *Q$* for an *N* character (lines 7070 through 7084) has to do with the "forgotten rule." All of this adds a bit of amusement and overcomes the problem of having too long a description for one page on a sixteen-line CRT. The reason an *N* is tested for: if the player had said yes when asked whether the description should be printed this trick will fetch the rest of it. A glance at line 7082 shows that if used it can only happen once, because *Q$* is overwritten with an *N* to forget forever the forgotten rule.

Red or Black

Back to *B$*. Line 7090 accepts the color choice, and lines 7100 through 7140 insure that either *nothing*, a *B*, or an *R* is typed. If nothing was typed, the numeric variable *B* is loaded with a nine; if black, the *B* code is a one; if red, *B* is made a zero.

Odd or Even

The choice about odd or even is next, beginning in line 7200. This is procedurally like the previous option. The typed response goes into *C$* and the numeric *C* is loaded with a nine for pass a one for odd, or a zero for even. One feature that was added to this option after watching others play *Roulette* for awhile is in line 7235. Too often novice operators seem confused about the letter O and the number zero. This trick eliminates any quandary. Either character will be accepted, and interpreted to mean O for odd. You and I and the computer know the difference between *0* and *O*; but why bore others?

High or Low

The fourth option—high or low—is also a redundant scheme. The prompt is in line 7320, and the input is followed by IF-THEN statements that make explicit tests for a *blank*, an *H* or an *L*. *D$* is the alpha variable this time, and numeric *D* is coded with a nine to

pass, a one for high, or a 0 for low. As in the preceeding options, if the explicit tests fail, a GOSUB 7700 is activated to sound the alarm.

Error Inputs

Each time a mistake is made, one of the smart remarks from the DATA strings in lines 7700 through 7720 is printed. This subroutine works as a perpetual READ loop. The trailing asterisk character denotes the end of the list; when encountered a final message is output and a *restore* is used to restart the list. It is funny, in play. The list is compiled so as to imply that the computer's patience is being sorely tried. Back to the norm.

The Wager

All that remains of player input is to "place your bet." Line 7430 asks for a dollar amount, which will be received into *X* after the receiver is set to zero (in line 7435). Nearly anything is tolerable at this point. A negative entry would not make much sense; so line 7460 will say "IMPOSSIBLE" if that ever happens. A pass or a typed-in zero at this point could mean the player wishes to quit playing. Line 7490 attempts a bit of intimidation, and asks if the player really does want to stop. If so, following the machine's retort (Thank Goodness), a branch is taken to line 3000 to wrap up the game. But maybe the zero was unintentional: THEN BET SOME MONEY.

A single bet of over $10,000 is considered unreasonable but anything less will allow a RETURN. Time out now. The markers are down, and the bet is made, so... roll it!

Wheel Spinner (8000—8350)

A pair of concentric loops are used at the start of this routine to simulate a wheel, coasting to a stop. A period character is printed between the two loops (with a leading space). The outside loop (lines 8010 through 8050) executes ten times, and the inside loop (lines 8020 through 8040) steadily increases in duration by a factor of ten times, the iteration count then in *I* (the outside loop's control variable). In execution this does appear something like a wheel coasting to a stop.

In line 8060 the wheel has stopped, and *A1* is loaded with a random number that supposedly shows where the ball has landed. To heighten the realism a final space, a period, and the *A1* value are displayed, all seemingly as a continuation of the wheel spinner loops.

The result in *A1* is tested for whether it is odd or even. This is done by lines 8100 through 8110, and 8240 through 8250. Depend-

ing on the number's parity, either a one or a 0 is loaded into *C1* (coinciding with *C*, which is where the player's code is stored).

Using a very similar technique, but dividing by three, the wheel's number is checked for red or black. The code of one or zero is again used, but this time it is placed in *B1* by line 8140 or 8260. Again there is a correspondence: the player's marker is in *B*.

The rest of the 8000-series coding does the printed output of the color, the parity, and whether the number is high or low. Notice that there is one exception—since zero cannot be divided, it is checked for explicitly; when it occurs the single PRINT in line 8280 does it all.

Read The Wheel (6000—6210)

A rather rote scheme does the comparisons of what the player chose with what the wheel spinner wrought. Each of the fields, *A* through *D*, are compared to their counterparts (*A1* through *AD1*), and for each matching pair a calculated score is placed into *M(1)* through *M(4)*. Notice that in lines 6000, 6040, 6090, and 6140 the *pass* trigger (ninety-nine or nine) is tested for; if found, that option is not qualified any further. (This works because *table M* is cleared back in the mainline prior to each round of play.)

Lines 6190 through 6196 include a table tally that totals into M(5), and line 6200 outputs the score for this spin and the money that has been accumulated thus far—whether it's your's or the machine's.

THE PROGRAM

```
 10 REM  "ROULETTE"
 20 REM
 30 GOSUB 9000
 35 DIM M(5)
 40 PRINT "WANT A DESCRIPTION (Y OR N)";
 50 INPUT @$
 60 IF @$ = "N" THEN 225
 70 IF @$ = "Y" THEN 110
 80 PRINT "PAY ATTENTION, I SAID (Y OR N)"
 90 PRINT
100 GOTO 40
110 PRINT "ROULETTE GAME"
120 PRINT "  BET ON A NUMBER (0-36)"
130 PRINT "  BET ON A COLOR (RED/BLACK)"
140 PRINT "  BET NUMBER IS ODD OR EVEN"
150 PRINT "  BET NUMBER IS LOW OR HIGH"
```

```
160 PRINT "      (0-18, 19-36)
170 PRINT "RESULTS OF BETTING ARE:"
175 PRINT
180 PRINT "..NUMBER.....WIN.....LOSE.."
190 PRINT "  NUMBER     BET X 5    BET"
200 PRINT "  COLOR      BET X 3  BET X 2"
210 PRINT " ODD/EVEN    BET X 2    BET"
220 PRINT " HIGH/LOW      BET      BET"
225 LET M = 0
230 PRINT
232 FOR I = 1 TO 5
234 LET M(I) = 0
236 NEXT I
240 GOSUB 7000
250 GOSUB 8000
260 GOSUB 6000
270 GOTO 230
3000 PRINT "END OF GAME, ";
3010 IF M < 0 THEN 3060
3020 IF M > 0 THEN 3080
3030 PRINT "WE'RE EVEN."
3040 PRINT
3050 END
3060 PRINT "PAY THE HOUSE $"ABS(M)
3070 GOTO 3040
3080 PRINT "YOU'RE LUCKY, COLLECT $"ABS(M)
3090 GOTO 3040
5999 REM  "READ THE WHEEL"
6000 IF A = 99 THEN 6040
6005 IF A <> A1 THEN 6030
6010 LET M(1) = X*5
6020 GOTO 6040
6030 LET M(1) = 0-X
6040 IF B = 9 THEN 6090
6050 IF B <> B1 THEN 6080
6060 LET M1 = M1+3*X
6070 GOTO 6090
6080 LET M(2) = 0-2*X
6090 IF C = 9 THEN 6140
6100 IF C <> C1 THEN 6130
6110 LET M(3) = 2*X
6120 GOTO 6140
6130 LET M(3) = 0-X
```

```
6140 IF D = 9 THEN 6190
6150 IF D <> D1 THEN 6180
6160 LET M(4) = X
6170 GOTO 6190
6180 LET M(4) = 0-X
6190 FOR I = 1 TO 4
6192 LET M(5) = M(5)+M(I)
6194 NEXT I
6196 LET M = M+M(5)
6200 PRINT "THIS TURN $"M(5)" TOTAL $"M
6210 RETURN
6999 REM  "PLAYER INPUT"
7000 PRINT "NUMBER (0-36)";
7005 LET A = 99
7010 INPUT A
7020 IF A < 0 THEN 7040
7025 IF A = 99 THEN 7060
7030 IF A < 37 THEN 7060
7040 GOSUB 7700
7050 GOTO 7000
7060 PRINT "BLACK/RED (B OR R)";
7065 LET B$ = " "
7070 IF Q$ = "N" THEN 7090
7075 PRINT
7080 GOSUB 7900
7082 LET Q$ = "N"
7084 GOTO7060
7090 INPUT B$
7100 IF B$ = " " THEN 7150
7110 IF B$ = "R" THEN 7190
7120 IF B$ = "B" THEN 7170
7130 GOSUB 7700
7140 GOTO 7060
7150 LET B = 9
7160 GOTO 7200
7170 LET B = 1
7180 GOTO 7200
7190 LET B = 0
7200 PRINT "ODD/EVEN (O OR E)";
7205 LET C$ = " "
7210 INPUT C$
7220 IF C$ = " " THEN 7270
7230 IF C$ = "O" THEN 7290
```

```
7235 IF C$ = "O" THEN 7290
7240 IF C$ = "E" THEN 7310
7250 GOSUB 7700
7260 GOTO 7200
7270 LET C = 9
7280 GOTO 7320
7290 LET C = 1
7300 GOTO 7320
7310 LET C = 0
7320 PRINT "HIGH/LOW (H OR L)";
7325 LET D$ = " "
7327 INPUT D$
7330 IF D$ = " " THEN 7380
7340 IF D$ = "H" THEN 7400
7350 IF D$ = "L" THEN 7420
7360 GOSUB 7700
7370 GOTO 7320
7380 LET D = 9
7390 GOTO 7430
7400 LET D = 1
7410 GOTO 7430
7420 LET D = 0
7430 PRINT "BET  $";
7435 LET X = 0
7440 INPUT X
7450 IF X => 0 THEN 7480
7460 PRINT "IMPOSSIBLE"
7470 GOTO 7430
7480 IF X <> 0 THEN 7560
7490 PRINT "COWARD...WANT TO QUIT (Y OR N)"
7500 INPUT J$
7510 IF J$ = "N" THEN 7540
7520 PRINT "THANK GOODNESS"
7530 GOTO 3000
7540 PRINT "THEN BET SOME MONEY"
7550 GOTO 7430
7560 IF X < 10000 THEN 7590
7570 PRINT "LET'S BE REASONABLE"
7580 GOTO 7430
7590 RETURN
7699 REM  "ERRORS"
7700 DATA OOPS, CAREFUL, HEY!, PLEASE!!
7710 DATA FUN-N-Y, HEADS-UP, ONCE-MORE
```

```
7720 DATA STOP-IT, LAST-CHANCE, *
7730 READ X$
7740 IF X$ <> "*" THEN 7790
7750 PRINT "YOU'VE EXHAUSTED MY PATIENCE.."
7760 PRINT "   NOW PLAY RIGHT!"
7770 RESTORE
7780 RETURN
7790 PRINT X$
7800 RETURN
7899 REM  "FORGOTTEN RULE"
7900 PRINT "I FORGOT TO EXPLAIN: RED NUMBERS"
7910 PRINT "  ARE ANY EVENLY DIVISIBLE BY 3"
7920 PRINT "  AND, TO SKIP ANY ENTRY DON'T"
7930 PRINT "  ENTER ANYTHING."
7940 PRINT
7950 RETURN
7999 REM  "WHEEL SPINNER"
8000 PRINT "SPINNING .";
8010 FOR I = 1 TO 10
8020 FOR J = 1 TO I*10
8030 NEXT J
8040 PRINT " .";
8050 NEXT I
8055 PRINT
8060 LET A1 = INT(100*RND(1))
8070 IF A1 > 36 THEN 8060
8080 IF A1 = 0 THEN 8280
8090 PRINT " ."A1"  ";
8100 LET C1 = INT(A1/2)
8110 IF A1-C1*2 = 0 THEN 8240
8115 LET C1 = 1
8120 LET B1 = INT(A1/3)
8130 IF A1-B1*3 = 0 THEN 8260
8140 LET B1 = 0
8150 IF B1 <> 0 THEN 8200
8160 PRINT "  BLACK";
8170 IF C1 <> 0 THEN 8220
8180 PRINT "  EVEN";
8190 GOTO 8330
8200 PRINT "  RED";
8210 GOTO 8170
8220 PRINT "  ODD";
8230 GOTO 8330
```

```
8240 LET C1 = 0
8250 GOTO 8120
8260 LET B1 = 1
8270 GOTO 8150
8280 PRINT " . 0  BLACK  EVEN  LOW"
8290 LET B1 = 0
8300 LET C1 = 0
8310 LET D1 = 0
8320 RETURN
8330 IF A1 < 19 THEN 8346
8340 LET D1 = 1
8342 PRINT "  HIGH"
8344 RETURN
8346 PRINT "  LOW"
8348 LET D1 = 0
8350 RETURN
9000 REM  "RANDOM NUMBER ROUTINE"
9010 LET Z = RND(1)
9020 RETURN
```

S

This game was stimulated by a ten year-old with an intense dislike for geography. He was supposed to learn the names of the states and their relative placements, and he liked to play games on the computer. So . . . .

The *States* program holds in memory the two-letter abbreviations of all fifty names. These are the same abbreviations that are acceptable to the post office. The complete list can be found in nearly any almanac for recent years.

The program also knows a clue about each state. There are ten clues in all, and for the most part they have a boundary or border connotation. For example, Alabama (AL) has a shoreline on the Gulf of Mexico. If the clue that is printed is GULF, a player might guess AL. If the guess is correct he or she then "owns" that state; if not, he or she simply loses the turn to the second player, who may then guess which other state it is that the computer is holding.

As the players watch each other's guesses one should eventually score. The program then moves the state to that player's account, and picks another from the shrinking list. The first player to own the names of ten states wins the game.

Although there are fifty states, the ten clues are not evenly distributed at the rate of five states per clue. One of the clues is ATLANTIC. Fourteen different states will prompt that clue. At the other end of the scale, only one state (WV) will trigger CHARLESTON clue. The ten clues and their relative distributions are as follows.

PACIFIC 3
ATLANTIC 14
CANADA 5
MEXICO 3
GULF 4
LAKES 6
REMOTE 2
RIVER 5
WESTERN 7
CHARLESTON 1

Variety abounds in this business, and the clues and their actual assignments were all my own whims. You may agree, or you may wish to coin other clues or assign them differently. The design of the *States* program makes it easy to vary these things—handy when some players get most of them down pat.

In fact, an advantage of having this program in your library is the variety of games that can be spun off of it. Other lists—the names of the months, astrological signs, psalms, and so on—can be easily overlayed with their own sets of clues. All of the driving mechanics of the program are insensitive to the names that are being guessed.

There is value offered by this program for other reasons also. The method used for accepting player inputs can be useful in a variety of programming problems. The method that is used is described best in conjunction with a look at the overall structure of the *States* program.

THE STATES TEMPLATE

For the most part our usual architecture prevails in this program. Looking at Fig. S-1 there is a bit of difference on the far right. It has to do with keyboard entries from the players. As announced in the game's instructions, whenever the program stops to allow the player to enter something from the keyboard, two optional requests may be made.

If the player types the word ALL, the program will print out all of the states. The remaining master list is shown as well as each of the two account files. After the lists are printed the program recovers to the entry point where it was when the ALL request was made.

The other option that is possible is to end the program. To do so the player simply types END. The program will acknowledge the request with a personalized signoff message, then terminate.

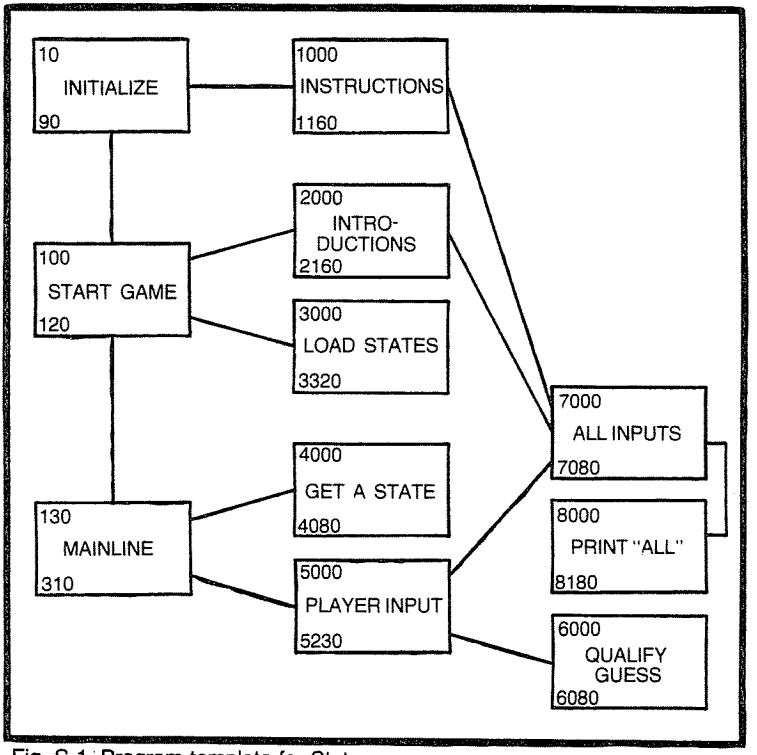


Fig. S-1. Program template for States.

As indicated by the template these requests may be made (1) after the instructions are printed, (2) while the players are introducing themselves to the computer, or (3) at any time while the game is in progress. All of which depends on the design technique of jumping to a common module to accept keyboard entries. There the keywords ALL and END are tested for and acted upon, or whatever else is entered is passed back to the calling sequence.

In a rather primitive way this design scheme is comparable to that used by system programmers for detecting and dispatching on "system commands." In concept at least this is how your system works when you command it to run, list, save, edit, or whatever. In some other areas in the *States* program I have borrowed from system programming methods also. Those tricks can be seen in the program's internals.

STATES INTERNALLY

Beginning in line 40 the program listing declares several work areas. The large alphanumeric table, *S$*, is sixty elements deep.

This table is set aside for the fifty names, plus the ten clue words. The two ten-element tables known as *P1$* and *P2$* are the "account files." As each player correctly identifies the name of a state its abbreviation is moved into that player's own table.

In line 50 there is another large table (*C*), used to hold fifty code numbers. This table will be looked at again a little later, where it will be seen to correspond with the fifty states' names; one *C* field being used for each state's clue code.

The remaining pair of workers are labeled *X*. By use of table-subscripting techniques the players' scores can be added into these two workers while within a looping sequence. More about that later.

The two GOSUB statements in lines 100 and 110 are for setting up the game. The first will jump to line 2000 for the introductions and the second to line 3000 for building the master list with the names of the states.

During the players' self-introductions the first use of the common inputs module can be seen. At the point in the dialog that their names are supposed to be typed (lines 2030 and 2050) the program does a jump to line 7000. Whatever is typed into *Q$* there is returned here for movement to the players' own worker, either *P1$* or *P2$*. If the data that is brought back is a space character, either that is what was typed, or the ALL option was exercised. Either way an empty *Q$* is useless; so line 2034 or 2054 will again ask for a player's name.

Lines 2060 through 2100 take care of making sure that the players' nicknames are alphabetic, at least in the first character position, and that the two names are visually different. If they didn't look different you would not be able to tell one from the other; all printed player references are done on a strictly personal basis. This tends to prove that people do not always have to be reduced to numbers just for the benefit of computers.

The rest of the subroutine that does the introductions business is also responsible for choosing which player gets to go first. This is done by the brief random-number sequence from line 2110 to line 2160. A single random integer is looked at, and *P* (the player control variable) is loaded with either a one or a two. If the number that is fetched is less than five the player that is known internally as number 1 will get to go first. If the random number is larger than five player 2 will be first up.

The other part of setting up a game has to do with loading the states' names and their clues into tables. The DATA series that begins in line 3000 includes each state's abbreviation, followed immediately by its corresponding clue code. The last ten DATA

elements are the clues themselves. Looking at this arrangement of DATA and the logic that follows it can be seen that anything could be defined here, allowing for other than the names of the states.

The RESTORE in line 3200 is to condition the READ pointer to the beginning of the list. The loop from line 3210 to line 3230 reads the first 100 DATA values, two at a time, and places the names in *S$* and the numbers in *table C*. Because the *I* variable is used concurrently the names and the codes will have a one-to-one correspondence in the two tables.

Immediately following this loop is another. When the first is completed the READ pointer should be established at the first clue word. The FOR expression in line 3240 is begun with *51* so that the clues will begin loading into the *S$* table from locations *51 through 60*. The reason for using two loops is, of course, because there are no code numbers needed after the clues. The first loop reads two values. The second reads only one at a time.

From lines 3270 through 3300 another loop is used. It runs ten times and blanks out the two account files. The final act in game initialization is in line 3310. Worker *H$* is for holding the name of the state that the program picks out at random. The reason that it must be space-filled is that conditional calls are made to "GET A STATE". The jump to line 3000 is only made whenever *H$* becomes empty.

All of the lines from 4000 to 4030 are involved in picking a random number of from one to fifty. This value is used as a subscript in line 4040 to load *H$* with a state's abbreviation (from *table S$*). The same value is used to pick up the corresponding clue code from *table C*, which is then used to obtain the clue itself.

When the clue code is obtained, if it is a negative number, the conditional in line 4050 causes a different state to be picked. If the code is negative it got that way during the course of play—when a player wins a state that state's code number is replaced by a negative form of the player's number. This works to mark the state as owned; yet it can still be printed in the ALL mode. The code switching business is done in player input (lines 5000 through 5230).

One of the player's names is printed by either line 5010 or 5020, depending on the content of *P*, which should have either a one or a two in it. Then, out of line 5040, a jump is made to line 7000 to permit the player to type a two-letter state name. The space test in line 5050 is similar to the one described before: if a blank is returned the same player is again asked for input.

Line 5060 checks if the player's guess is correct. If it is line 5070 replaces the clue code. This trick is often used in the design of

compilers and assemblers. The number in its absolute form is still useful, but marking it with a positive or negative sign makes it possible to have distinguishing characteristics—without otherwise corrupting it.

The series from line 5080 to 5140 is a FOR-NEXT loop that is used to move the state to the next available slot in the player's account file. The series from line 5170 to line 5190 is an extension of the same routine, needed to act on *table P2$* instead of *P1$*. When working in either area, if the FOR-NEXT loop expires by running a full ten times the added message "AND YOU WIN" follows the printing of the player's current score.

Line 5200 is the point at which execution is branched to if the player's guess does not match the state being held in *H$*. An immediate jump is made to line 600 to qualify whatever it was that was typed into *Q$*.

The first thing to be checked for is whether the input guess is in fact the correct abbreviation for some state. A FOR-NEXT loop lines (6000 through 6020) is used to scan down *table S$*, comparing *Q$* with each name, one at a time. If the loop ever does run beyond fifty times with no match *Q$* must be bad. If it is *Q$* is cleared; and when the program returns to the calling sequence the player will be allowed to guess again.

Another possibility is that the abbreviation is a valid one but one that has already been played and won by one of the two players. This is where that negative code comes in handy, again. Line 6030 looks for this condition; if found, line 6070 prints an "ALREADY OWNED" message. Like the "INVALID" one *Q$* is sent back empty to let the player try again.

The remaining routine is that called *print all*. The only jump to line 8000 is that coming from line 7030 (in the *all inputs* routine); yet this subroutine expects to be called upon nearly any time.

This is where finally the two-element table called *X* is used. As the loop from line 8020 to line 8060 runs, whenever a state's corresponding code field is negative, that state is temporarily skipped during the printout. As the mass printing is going on, however, the absolute value of the code (one or two) serves as a subscript to increment one of the *X* counters.

Two more loops are used then, conditionally, to print out the states that are marked as belonging to each of the players. Both loops are set to run according to the *X* counter, and the names themselves are obtained from the respective player's file.

By now you will have noticed that it really doesn't matter what it is that is stored in *table S$*. Taken just as is *States* can be fun and

educational; and the program's design can be used for more than just for geography. However it is used it is worthwhile—it can make learning fun—for children and adults.

THE PROGRAM

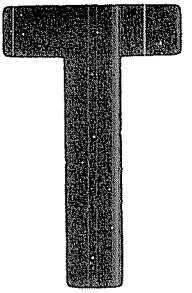
```
 10 REM  "STATES"
 20 REM
 30 GOSUB 9000
 40 DIM S$(60), P1$(10), P2$(10)
 50 DIM C(50), X(2)
 60 PRINT "WANT INSTRUCTIONS (Y OR N)";
 70 INPUT Q$
 80 IF Q$ = "N" THEN 100
 90 GOSUB 1000
100 GOSUB 2000
110 GOSUB 3000
120 PRINT
130 IF H$ <> " " THEN 150
140 GOSUB 4000
150 IF P = 1 THEN 200
160 GOSUB 5000
170 LET P = 1
180 IF I = 10 THEN 230
190 GOTO 130
200 GOSUB 5000
210 LET P = 2
220 GOTO 180
230 PRINT "ANOTHER GAME (Y OR N)";
240 INPUT Q$
250 IF Q$ = "Y" THEN 280
260 PRINT "GOOD-BYE "P1$", AND "P2$
270 END
280 PRINT "SAME PLAYERS (Y OR N)";
290 INPUT Q$
300 IF Q$ = "Y" THEN 110
310 GOTO 100
1000 PRINT "I KNOW ALL 50 STATES BY THEIR"
1010 PRINT "2-LETTER ABBREVIATIONS.  I'LL"
1020 PRINT "PICK ONE AND GIVE YOU A CLUE."
1030 PRINT "YOU TAKE TURNS TRYING TO GUESS"
1040 PRINT "THE STATE.  THE FIRST PLAYER TO"
1050 PRINT "'OWN' 10 STATES WINS."
1060 PRINT
```

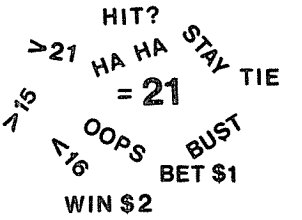
```
1070 PRINT "FOR A LIST OF THE STATES TYPE"
1080 PRINT "  'ALL'  (ANYTIME)."
1090 PRINT
1100 PRINT "TO END THE GAME TYPE"
1110 PRINT "  'END'  (ANYTIME)."
1120 PRINT
1130 PRINT "READY";
1140 GOSUB 7000
1150 PRINT
1160 RETURN
1999 REM  "INTRODUCTIONS"
2000 LET M1$ = "PLAYER #"
2010 LET M2$ = ", WHAT IS YOUR NAME"
2020 PRINT M1$"1"M2$
2030 GOSUB 7000
2032 LET P1$ = Q$
2034 IF Q$ = " " THEN 2020
2040 PRINT M1$"2"M2$
2050 GOSUB 7000
2052 LET P2$ = Q$
2054 IF Q$ = " " THEN 2040
2060 IF P1$ < "A" THEN 2090
2070 IF P2$ < "A" THEN 2090
2080 IF P1$ <> P2$ THEN 2110
2090 PRINT "PLEASE TRY AGAIN."
2100 GOTO 2020
2110 LET P = INT(10*RND(1))
2120 IF P < 5 THEN 2150
2130 LET P = 2
2140 RETURN
2150 LET P = 1
2160 RETURN
2999 REM  "LOAD STATES"
3000 DATA AL,5,AK,7,AZ,4,AR,8,CA,1
3010 DATA CO,9,CT,2,DE,2,FL,5,GA,2
3020 DATA HI,7,ID,3,IL,6,IN,6,IA,8
3030 DATA KS,9,KY,8,LA,5,ME,2,MD,2
3040 DATA MA,2,MI,6,MN,3,MS,5,MO,8
3050 DATA MT,3,NE,9,NV,2,NH,2,NJ,2
3060 DATA NM,4,NY,2,NC,2,ND,3,OH,6
3070 DATA OK,9,OR,1,PA,6,RI,2,SC,2
3080 DATA SD,9,TN,8,TX,4,UT,9,VT,3
3090 DATA VA,2,WA,1,WV,10,WI,6,WY,9
```

```
3100 DATA PACIFIC, ATLANTIC, CANADA
3110 DATA MEXICO, GULF, LAKES, REMOTE
3120 DATA RIVER, WESTERN, CHARLESTON
3200 RESTORE
3210 FOR I = 1 TO 50
3220 READ S$(I), C(I)
3230 NEXT I
3240 FOR I = 51 TO 60
3250 READ S$(I)
3260 NEXT I
3270 FOR I = 1 TO 10
3280 LET P1$(I) = " "
3290 LET P2$(I) = " "
3300 NEXT I
3310 LET H$ = " "
3315 LET I = 0
3320 RETURN
3999 REM  "GET A STATE"
4000 LET H = INT(100*RND(1))+1
4010 IF H =< 50 THEN 4040
4020 LET H = INT(H/2)
4030 IF H < 1 THEN 4000
4040 LET H$ = S$(H)
4050 IF C(H) < 0 THEN 4000
4060 LET X = C(H)
4070 LET C$ = S$(X+50)
4080 PRINT C$
4090 RETURN
4999 REM  "PLAYER INPUT"
5000 IF P = 1 THEN 5030
5010 PRINT P2$;
5020 GOTO 5040
5030 PRINT P1$;
5040 GOSUB 7000
5050 IF Q$ = " " THEN 5000
5060 IF Q$ <> H$ THEN 5200
5070 LET C(H) = 0-P
5080 FOR I = 1 TO 10
5090 IF P = 1 THEN 5170
5100 IF P2$(I) <> " " THEN 5140
5110 LET P2$(I) = H$
5170 PRINT "YOU NOW OWN"I
5174 LET H$ = " "
```

```
5125 IF I = 10 THEN 5150
5130 RETURN
5140 NEXT I
5150 PRINT "AND YOU WIN"
5160 RETURN
5170 IF P1$(I) <> " " THEN 5140
5180 LET P1$(I) = H$
5190 GOTO 5120
5200 GOSUB 6000
5210 IF Q$ = " " THEN 5000
5220 PRINT "SORRY"
5230 RETURN
5999 REM  "QUALIFY GUESS"
6000 FOR I = 1 TO 50
6010 IF Q$ = S$(I) THEN 6030
6020 NEXT I
6025 GOTO 6040
6030 IF C(I) < 0 THEN 6070
6035 GOTO 6055
6040 PRINT "INVALID"
6050 LET Q$ = " "
6055 LET I = 0
6060 RETURN
6070 PRINT "ALREADY OWNED"
6080 GOTO 6050
6999 REM  "ALL INPUTS"
7000 LET Q$ = " "
7010 INPUT Q$
7020 IF Q$ <> "ALL" THEN 7060
7030 GOSUB 8000
7040 GOTO 7000
7050 RETURN
7060 IF Q$ <> "END" THEN 7050
7070 PRINT "SO LONG "P1$"; AND "P2$
7080 END
7999 REM  "PRINT ALL"
8000 LET X(1) = 0
8010 LET X(2) = 0
8015 LET J = 0
8020 FOR I = 1 TO 50
8030 IF C(I) > 0 THEN 8050
8040 LET X(ABS(C(I))) = X(ABS(C(I)))+1
8045 GOTO 8060
```

```
8050 PRINT S$(I)" ";
8052 LET J = J+1
8054 IF J < 11 THEN 8060
8056 PRINT
8058 LET J = 0
8060 NEXT I
8070 IF X(1) = 0 THEN 8120
8074 PRINT
8076 PRINT
8080 PRINT P1$ " HAS"X(1)
8090 FOR I = 1 TO X(1)
8100 PRINT P1$(I)" ";
8110 NEXT I
8120 IF X(2) = 0 THEN 8170
8124 PRINT
8126 PRINT
8130 PRINT P2$" HAS"X(2)
8140 FOR I = 1 TO X(2)
8150 PRINT P2$(I)" ";
8160 NEXT I
8170 PRINT
8180 RETURN
9000 REM  "RANDOM NUMBER ROUTINE"
9010 LET Z = RND(1)
9020 RETURN
```





Twenty1

Nearly everyone has played this card game, which may be better known by its other name, blackjack. By either name, so too, has nearly every programmer written at least one program to play it. Most of us have programmed it several times, several different ways, and perhaps in different languages. Here is yet another.

The first time that you do it the game's mechanics provide an interesting programming challenge. After that it is sometimes more of a challenge to find *something else challenging about it* rather than the rote mechanics of dealing the cards and tallying their worth. That can become old hat in short order.

My enthusiasm for this iteration was finally rekindled by the challenge of conceiving a new card-deck management scheme. Which does tend to prove that, no matter how tiring some problems seem, if we ponder long enough, the fun of designing novel solutions to them can still be had.

THE DECK

There are four suits in a regular deck of playing cards. Without any special regards for which is which (in this game it doesn't matter), each suit is assigned one of the numbers one, two, three, or four. For graphic purposes, the assignments used here are, in order: hearts, spades, diamonds, and clubs.

Each suit has thirteen cards. There is an ace, the numbers 2 through 10, a jack, a queen, and a king. In contrast to the suits which have a code number assigned to them physically the individual card's

values are logically assigned. This is achieved by using a table that is thirteen elements deep, each element being reserved to hold four suit codes.

The first position in the table represents all four aces. The next position down is for the twos, the next is for the threes, and so on. Notice that the card values themselves can be derived from the table element's position rather than from an actual code number.

The dealing mechanism uses a random number (between one and thirteen) to look into the table at a relative position (from the top). What it may find there is *1234*. Or it may find *4321*, or *2314*, or *3124*, or something similar.

When a new deck is built the suit codes are randomly jumbled, one code for each suit all packed together, and the resulting combinations are moved into the table, one line at a time. Each line's combination is individually scrambled, but each set is comprised of the four digits 1,2,3, and 4.

During the course of play, as the cards are dealt, the suit codes are removed from the table one at a time. The algorithm that does the dealing first picks a random number of from one to thirteen. That number is then used as a subscript to access an element of the table. Next, the low-order (rightmost) digit of the code set is removed and the remaining digits are shifted to the right by one position. The shortened result is put back into the table at the same location from which it was taken.

The number that was removed (shifted out) is used in conjunction with the accessing subscript to represent the card then being dealt. To get the alphanumeric PRINT values, a DATA structure is used, and a READ to obtain the corresponding card's name. The suit's name is obtained by use of it's code, picking up the name from a four-field table of alphanumeric constants. The exact mechanics of decoding the cards will be seen more clearly in the program's listing. Before that, however, there is an aspect of this deck-storage scheme that deserves recognition.

A DESIGN AUDIT

Remembering that the four codes that denote the suits are removed, one at a time from each field in the deck, eventually the elements are zeroed out. The subroutine that randomly selects one of the thirteen fields is constructed so that if one element is empty another must be looked at. Obviously, as the deck is used up, more and more of the elements have all zeroes — thus, the longer it takes to find a number. When there is but one code left, in only one of the elements (meaning fifty one cards have been dealt), it can take a while to find that last card.

Whether the length of time is too long or not is a matter of judgment. In my experience the wait has been quite acceptable. Before the design was fully committed, however, I used an audit mechanism that can be used in a variety of situations.

A special-purpose variable is allocated for saving the loop count at the point a card code is found. Beginning with a newly constructed deck the very first attempt has to be valid so the auditor will contain a one. From then on, continuing as long as the program is active, every time the dealer subroutine is executed, if the loop count exceeds that previously attained, the auditor is updated with this higher number.

The following logic does this, using *A8* as a temporary counter and *A9* as the auditor.

```
LET A8 = A8+1 (increment the counter, each loop)

IF A9 > A8 THEN (skip a line)

LET A9 = A8 (overwrites the auditor)
```

Additionally, it is necessary to initialize the *A8* temporary counter with a zero somewhere preparatory to the jump to the card-dealing sequence.

The history that is achieved in the auditor ought to be looked at periodically. Preference can prevail as to how. I included a *PRINT A 9* inline immediately after it was updated; but you could instead merely stop the program from time to time to look at the auditor.

However it is employed, this trick, borrowed from "systems tuning" concepts, is handy in many ways. (By the way, my biggest count to date was 204. Only rarely does it ever broach 100, however.)

PUTTING TWENTY1 TOGETHER

The layout of the program's major pieces can be seen in Fig. T-1. As shown in the drawing, after the program is initialized and after the instructions option is allowed, the program proceeds to set up the game. The principal set up task is to create a deck of cards in memory. The drawing also shows that the new deck module may be called later if, as the cards are being dealt, the deck becomes depleted.

In the center of the template there is a major module for the human's play and another for the computer's play. At the start of a round these two tasks are called by a sequencer that cycles until

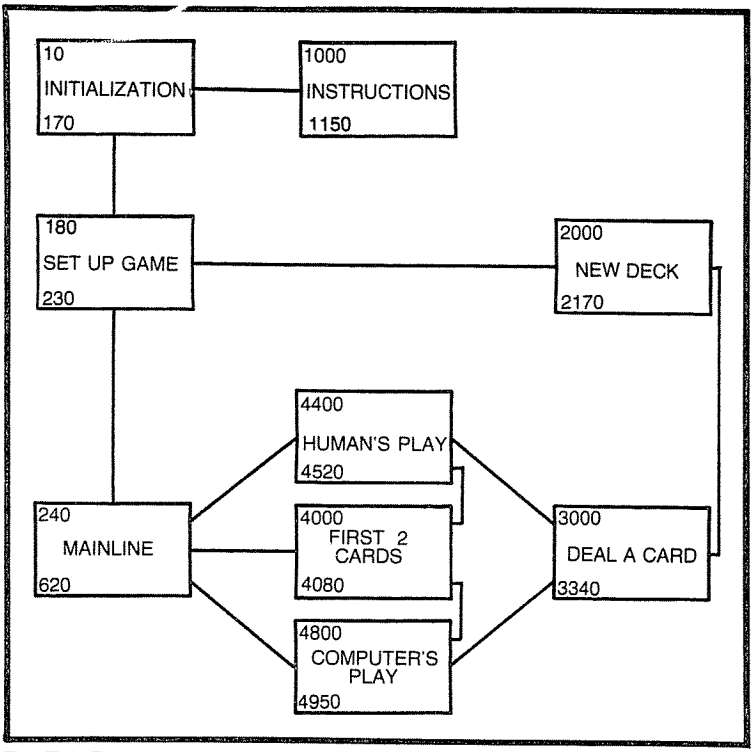


Fig. T-1. Program template for Twenty I.

both players have been dealt their first two cards. From there on the player modules are called directly from the mainline as they are needed.

There is only the one module that serves to deal the individual cards. An advantage of this layout is that if you want to experiment with other deck schemes only two modules have to be replaced: the one that creates the deck and the one to deal a card.

The program does work quite well just as presented, and whether you decide to copy it as is or not, its internal mechanics should be observed closely. The following descriptions are intended to assist you in your study of *Twenty1*.

Initialization (10—170)

The DIM statement in line 40 establishes three single-dimension tables. *Table D* has thirteen numeric elements for holding the deck. The four-element table called *S* is a work area for holding the suit codes (one, two, three, and four) during the process of shuffling them as a set, in conjunction with the deck-building sequence.

The remaining table *(S$)* is for holding the names of the suits. Lines 50 through 80 fill *table S$,* immediately and in the simplest way possible. The reason a table is used rather than simple variables is that it permits the dealer to use the suit codes as direct subscripts to obtain the printable names of the suits.

Presumably the instructions business (lines 90 through 180) needs no explanation. There is certainly nothing tricky there, nor in the series of print statements from line 1000 to 1140, which end with a return to line 180.

Set Up Game (180—230)

Symbol *M* if for me. *T* is for you. These are assigned from the computer's point of view, of course. These are the scorekeepers, and that is why they must be cleared before each new game is started. The other part of setting up a new game includes the jump out of line 220 to build the first deck of cards.

New Deck (2000—2170)

First, the suit numbers one, two, three, and four are loaded into *table S.* The short loop from line 2000 to line 2020 does this. A pair of concentric FOR-NEXT loops are used then to concurrently jumble the suit codes and assemble them into groups of four as they are placed into the deck table. The outside loop (controlled by *I)* manages the table insertions; it runs thirteen times.

The loop that is controlled by *J* extends from line 2040 down to line 2110. A random integer is used to exchange the codes in the second, third, or fourth element of *table S* with whatever is in the top spot. The exchange process is conditioned to run four times. Fewer than four shuffles did not seem to give much variety. Running more than four times slows up the whole works. Four was finally settled upon as sufficient but not excessive. It wouldn't be necessary to shuffle them at all, but some players of *Twenty1* might get suspicious if the suits appeared with a fixed regularity.

After the shuffle loop, lines 2120 through 2140 use simple arithmetic to combine the codes together into four-digit numbers. As a set then, they are placed directly into the deck at the location that is currently aimed at by the *I* variable. When the deck is all built, line 2160 sets the deck's counter *(D)* to zero, and the RETURN in line 2170 is executed.

Deal A Card (3000—3340)

Each time this routine is called line 3000 checks whether the deck has run out. If it has, before a new deck is created, the player is

asked whether he wants or she wants to stop. If the answer is yes the program terminates. Anything other than a *Y* will cause a jump to line 2000 (to build a new deck of cards), and the game continues.

Line 3090 is the one responsible for incrementing the deck's counter. This is the same continuation point that line 3000 branches to as each card is dealt and as long as the count has not yet reached fifty two.

Lines 3100 and 3110 fetch a random number (two digits); if the number that is handed up is larger than thirteen, line 3120 subtracts thirteen from it. The branch in 3130 causes the reduced amount to be checked, and the sequence repeats itself until a valid number is obtained (one to thirteen).

The card number that is now in *C* is used in line 3140 to remove a set of suit codes from the deck table. Before going any further line 3150 checks if the value in *C* is zero; if so, the pickup sequence is run through again. (The auditor routine that was described earlier will fit nicely here, between lines 3140 and 3150.)

The rightmost suit code is parsed from the set that was picked up. This is done by lines 3160, 3170, and 3180. The resulting single digit is now in *S*, and *S* serves as the subscript to load *S$* with that suit's name.

The DATA strings in lines 3200 and 3210 include the names of all of the cards in ascending sequence. The FOR-NEXT loop in lines 3220 through 3240 is set up to run for *C* times. *C* still has the value in it that was used to get the suit code out of the deck. When the READ loop stops *C$* will have in it the corresponding card's name.

At this point the card is available and ready to be delivered. Before it is output, however, *C* is adjusted (lines 3260 through 3270) so that the code will not be larger than ten for jacks and above, since ten is the point value of all face cards.

Now *C$* (the card) is printed along with *S$* (the suit), and the task is finished, with the card's point value in *C*.

First Two Cards (4000—4080)

At the start of a round of play each of the two hands (A1 and B1) are set to zero. So are the ace counters (A2 and *B2)*. The human is first: those counters are *A*. The dealer always plays last, so those counters are in the two *B* workers. Four jumps in sucession are used then, to alternately deal each two cards, beginning with the jump to line 4400 for the player's first card. The final return from here will be back to the mainline for the draw process.

Human's Play (4400—4520)

This is the first player, so, *P* is set to one in line 4400. Now GOSUB 3000 is invoked to get a card from the dealer. If the card that is

returned in *C* is not an ace (is not a one) life is simple. Aces can optionally count as one or an eleven. Most of this routine's complexity has to do with managing the aces.

The first test that is made (in line 4410) is to see if the card is an ace. If not, will this card increase the points in *A1* beyond twenty one? If not, simply add the card to the hand and exit lines (4430 through 4440).

If a "bust" is in the making (about to exceed twenty one the branch to line 4490 is to see whether any aces are already present in the hand, and if so, to change their worth from eleven to one, if possible. *A2* is this player's ace counter; if it is empty (checked in line 4490), the player is in trouble. There is nothing else this routine can do, however, so *A1* is increased anyway and the routine is exited.

The *A2* counter only keeps track of eleven point aces. If there is a number in *A2*, one is subtracted from it, and the hand is reduced by ten points (lines 4500 and 4510). Now, again, the previous try is made to see whether the card that is being dealt will go over twenty one.

Most of this logic is the same as when an ace is first dealt. In that case, however, it must be immediately determined whether the ace should be put away initially as a one or as an eleven. If eleven will fit, as checked for in line 4450, lines 4460 and 4470 will adjust *C* accordingly, and add one to the ace counter. If an eleven point ace will not fit, it is counted as a one, and treated no differently from any other card.

Computer's Play (4800—4950)

Whatever is fair for humans is fair for computers. Maybe that should be restated— the other way around. Anyway, this routine is nearly identical to the preceding one. The only difference is at the outset: if the dealer's hand (*B1*) has seventeen or more points in it, it gets no more cards. An immediate exit is taken (line 4820).

The Draw

Back in the mainline, after the players have received their first two cards, the human is asked if he or she would like a hit. The option is offered in line 310, but only if *A1* is less than twenty one. Blackjacks are detected automatically by line 280. In line 270 there is also a test to see whether both hands have twenty one already. In the event of a tie at the outset, the dealer wins and another round is started. This can be seen by tracking the branches from line 270.

If only *A1* has twenty one the dealer's draw must be gone through. The dealer might be able to match exactly on twenty one. The exit in this case (from line 280) is to line 510 to first print "BLACKJACK," then on to line 360 for the dealer's draw. This is the same point that the human's hit option will finally get to when finally, the player either chickens and says no, or goes bust by exceeding twenty one points.

The dealer's drawing mechanics are slightly more complex, since there is no benefit of a human's intelligence. The first decision is made in line 370. If *B1 is* sixteen or less the dealer must draw. In line 380 if the dealer's hand ever matches the other exactly, the dealer wins on the basis of the dealer-takes-pushes "tie" rule. The dealer will also win at any point that the computer's hand exceeds the other's (the default branch in line 415), but two other things can happen before that.

The dealer can go bust. This possibility is checked in line 390. The dealer can also get lucky and get a Blackjack, as tested for in line 400. If so, the dealer needs no further cards and the exit is down through line 420. All of those short two-and three-line sequences below are for outputting the appropriate messages, adjusting the scores, and routing the flow back to line 240 for another round. So the game goes on.

THE PROGRAM

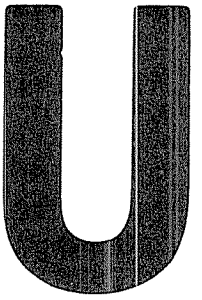
```
 10 REM  "TWENTY1"
 20 REM
 30 GOSUB 9000
 40 DIM D(13), S(4), S$(4)
 50 LET S$(1) = "HEARTS"
 60 LET S$(2) = "SPADES"
 70 LET S$(3) = "DIAMONDS"
 80 LET S$(4) = "CLUBS"
 90 PRINT
100 PRINT "WANT INSTRUCTIONS (Y OR N)";
110 INPUT Q$
120 IF Q$ = "Y" THEN 160
130 IF Q$ = "N" THEN 180
140 PRINT "HUH?"
150 GOTO 90
160 PRINT
170 GOSUB 1000
180 PRINT
190 LET M = 0
```

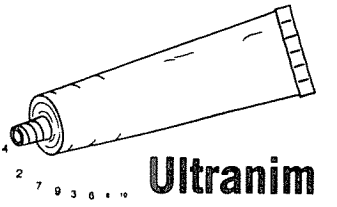
```
200 LET Y = 0
210 PRINT "NEW DECK"
220 GOSUB 2000
230 GOTO 250
240 PRINT "SCORES:  YOU $"Y"  ME $"M
250 PRINT
260 GOSUB 4000
270 IF A1+B1 = 42 THEN 490
280 IF A1 = 21 THEN 510
290 IF A1 > 21 THEN 470
300 LET Q$ = "Y"
310 PRINT "HIT";
320 INPUT Q$
330 IF Q$ = "N" THEN 360
340 GOSUB 4400
350 GOTO 270
360 GOSUB 4800
370 IF B1 < 17 THEN 360
380 IF B1 = A1 THEN 490
390 IF B1 > 21 THEN 530
400 IF B1 = 21 THEN 420
410 IF B1 < A1 THEN 540
415 GOTO 430
420 PRINT TAB(16) "BLACKJACK!"
430 LET Y = Y-1
440 PRINT TAB(16) "I WIN $1.00"
450 LET M = M+1
460 GOTO 240
470 PRINT "BUSTED, HA HA";
480 GOTO 430
490 PRINT TAB(10) "TIE";
500 GOTO 430
510 PRINT "BLACKJACK!  ";
520 GOTO 360
530 PRINT TAB(16) "OOPS"
540 IF A1 <> 21 THEN 590
550 PRINT "YOU WIN $2.00"
560 LET Y = Y+2
570 LET M = M-2
580 GOTO 240
590 LET Y = Y+1
600 LET M = M-1
610 PRINT "YOU WIN $1.00"
```

```
 620 GOTO 240
1000 PRINT "A BLACKJACK IS 21 POINTS."
1010 PRINT "  2 THROUGH 10 = FACE VALUE"
1020 PRINT "  FACE CARDS = 10 POINTS"
1030 PRINT "  ACES = 1 OR 11"
1040 PRINT "WE START WITH TWO CARDS APIECE."
1050 PRINT "I AM THE DEALER."
1060 PRINT "I ALWAYS STAND ON 17 OR MORE."
1070 PRINT "I MUST DRAW ON 16 OR LESS."
1080 PRINT "I WIN ALL TIES."
1090 PRINT "BETS ARE $1."
1100 PRINT "BLACKJACKS PAY DOUBLE."
1110 PRINT "A DECK HAS 52 CARDS."
1120 PRINT "THE GAME MAY BE ENDED WHEN THE"
1130 PRINT "DECK RUNS OUT, OR YOU MAY GO ON."
1140 PRINT "READY";
1150 INPUT Q$
1160 RETURN
1999 REM  "NEW DECK"
2000 FOR I = 1 TO 4
2010 LET S(I) = I
2020 NEXT I
2030 FOR I = 1 TO 13
2040 FOR J = 1 TO 4
2050 LET R = INT(10*RND(1))
2060 IF R < 2 THEN 2050
2070 LET R = INT(R/2)
2080 LET S = S(R)
2090 LET S(R) = S(1)
2100 LET S(1) = S
2110 NEXT J
2120 LET D(I) = S(1)+S(2)*10
2130 LET D(I) = D(I)+S(3)*100
2140 LET D(I) = D(I)+S(4)*1000
2150 NEXT I
2160 LET D = 0
2170 RETURN
2999 REM  "DEAL A CARD"
3000 IF D < 52 THEN 3090
3010 PRINT "END OF DECK"
3020 PRINT "WANT TO STOP (Y OR N)";
3030 INPUT Q$
3040 IF Q$ <> "Y" THEN 3070
```

```
3050 PRINT "SO LONG ..."
3060 END
3070 PRINT "NEW DECK"
3080 GOSUB 2000
3090 LET D = D+1
3100 LET C = INT(100*RND(1))+1
3110 IF C < 14 THEN 3140
3120 LET C = C-13
3130 GOTO 3110
3140 LET S = D(C)
3150 IF S = 0 THEN 3100
3160 LET J = INT(S/10)
3170 LET D(C) = J
3180 LET S = S-J*10
3190 LET S$ = S$(S)
3200 DATA ACE, 2, 3, 4, 5, 6, 7, 8, 9
3210 DATA 10, JACK, QUEEN, KING
3220 FOR I = 1 TO C
3230 READ C$
3240 NEXT I
3250 RESTORE
3260 IF C < 11 THEN 3280
3270 LET C = 10
3280 IF P <> 1 THEN 3330
3300 PRINT C$" "S$
3310 RETURN
3330 PRINT TAB(16) C$" "S$
3340 RETURN
3999 REM  "FIRST 2 CARDS"
4000 LET A1 = 0
4010 LET A2 = 0
4020 LET B1 = 0
4030 LET B2 = 0
4040 GOSUB 4400
4050 GOSUB 4800
4060 GOSUB 4400
4070 GOSUB 4800
4080 RETURN
4399 REM  "HUMAN'S PLAY"
4400 LET P = 1
4405 GOSUB 3000
4410 IF C = 1 THEN 4450
4420 IF A1+C > 21 THEN 4490
```

```
4430 LET A1 = A1+C
4440 RETURN
4450 IF A1+11 > 21 THEN 4490
4460 LET A2 = A2+1
4470 LET C = 11
4480 GOTO 4430
4490 IF A2 < 1 THEN 4430
4500 LET A2 = A2-1
4510 LET A1 = A1-10
4520 GOTO 4420
4799 REM  "COMPUTER'S PLAY"
4800 IF B1 < 17 THEN 4830
4820 RETURN
4830 LET P = 2
4835 GOSUB 3000
4840 IF C = 1 THEN 4880
4850 IF B1+C > 21 THEN 4920
4860 LET B1 = B1+C
4870 RETURN
4880 IF B1+11 > 21 THEN 4920
4890 LET B2 = B2+1
4900 LET C = 11
4910 GOTO 4860
4920 IF B2 < 1 THEN 4860
4930 LET B2 = B2-1
4940 LET B1 = B1-10
4950 GOTO 4850
9000 REM  "RANDOM NUMBER ROUTINE"
9010 LET Z = RND(1)
9020 RETURN
```





Some games involve you and the computer as opponents. Others enable you to challenge a friend, using the computer as nothing more than the ultimate in fancy game boards. The program presented in this chapter does either of these optionally or both at the same time.

The game itself is based on the ancient game of nim—probably the most programmed game in the world. Because you can play against the computer, against a friend, or the three of you may compete at the same time, this is called *Ultra*nim.

As a program. *Ultranim* contains all the techniques necessary to enable the computer to play a perfect game of nim. There is also within it programming tricks that can be used for the control of other games that involve two people and the machine.

This is how *Ultranim* presents itself. A game begins with five groups of five tokens each. In a turn a player may pick from one to all of the tokens in any groups. The player who gets the last pick wins the game.

STRATEGIES IN NIM

Classical versions of this game involve only two players, and the winner can be predicted exactly before the game even begins. This does assume that both players play a perfect game, of course. This is especially necessary of the one with the strategic advantage. Any error on his or her part can ;ill nullify his advantage. As a case in point, suppose the following:

- 100 tokens
- A draw cannot be less than one nor more than ten
- Player who gets the last draw is the winner.

In the above variation of nim the player who goes first has a real advantage and should always win. In your first turn you should take one token. Thereafter, the number that you should take should be such that your pick added to your opponent's will equal eleven every time. The results: 99, 88, 77, . . . 22, 11, and in the end, zero.

Ultranim offers several deviations to the classical scheme; yet it can still be predicted who ought to win in every case. There is the difference that there are only twenty-five tokens. They are grouped as sets of five, and a player may take any or all of the five in any one group in his or her turn; and either two or three players may compete, one of which may be the computer.

Without mapping out truth tables for the various combinations of players copy this program and play it for a while. I debugged it in just that way, and it was surprising how many games I had to play before the pattern could be discerned as to know how to beat the machine. Amusingly enough this is true even when we already know how the program makes its moves.

THE COMPUTER'S CHOICE

A game line is printed by the program like this:

```
••1••  ••2••  ••3••  ••4••  ••5••
XXXXX XXXXX XXXXX XXXXX XXXXX
```

A move choice by any of the players is then, a group number, followed by the pick within that group. Whenever it is the computer's turn, it first looks at the largest group. It does work from left to right; so in early rounds the first encountered with all five tokens still in place will be its group choice. As the groups are diminished during successive rounds of play the program will always look for the largest remaining group, choosing from the left in cases of equally sized groups. Then it does a thing involving binary numbers and parity checks.

The real numbers of 0, 1, 2, 3, 4, and 5 have the binary equivalents of 000, 001, 010, 011, 100, and 101. The program uses a table of these equivalents to convert the number of tokens remaining in each group to a binary value. An example is shown in Fig. U-1 of what a partially played out game will produce.

The binary equivalents are stacked one above the other and aligned, just as if arithmetic was about to be performed. That is in

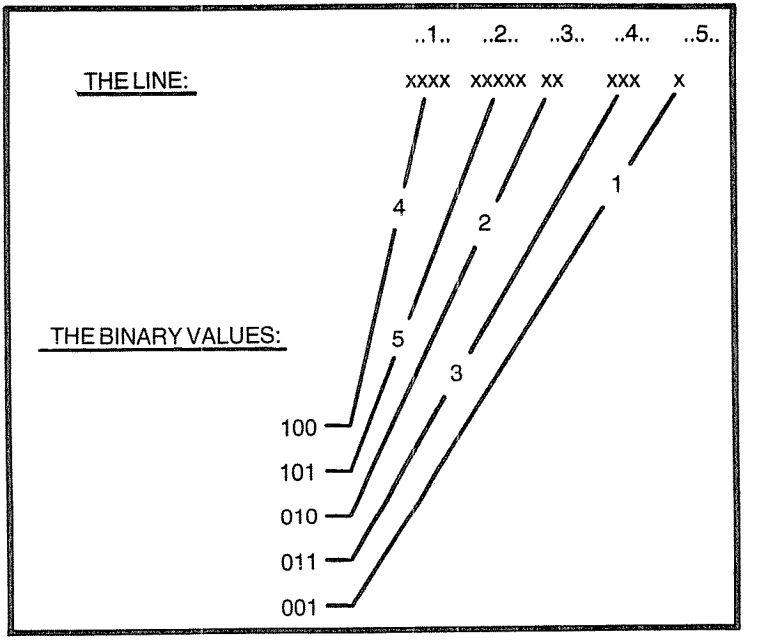


Fig. U-1. Number of Ultranim tokens per group, in binary.

fact what is done next. In Fig. U-2 the same set of binary numbers is used again, showing the result of addition down their vertical columns. This addition is done always without any consideration for carrying.

In effect what is achieved is a column-by-column parity value. If there is an odd number of ones in a column, a one will result in the total (in that columnar position). Conversely, if there is an even number of ones in a column, the result for that column will be zero.

The computer now makes its play, taking sufficient tokens from the largest group to bring the overall parity situation to all zeros. As implemented within the program this algorithm always works, regardless of the order of play. If, as it can happen parity exists at the outset(when it is the computer's turn) it always first removes one token, then it checks the parity. If that one pick produced the desired result the program relinquishes its turn; it took the minimum permitted. If, on the other hand, an odd parity is present after removing one token it takes another. And it checks the parity again. It takes another and another until an even parity situation is produced.

This scheme really does work to cause perfect play. When the computer's starting position gives it the strategic advantage it will

always win. It is also true that it will play a perfect loser, and that it will assume the advantage if allowed to do so.

THE REST OF ULTRANIM

All of the things that go on in *Ultranim* are pictorially represented in Fig. U-3. As usual there are instructions provided as the game is started the first time. Slightly unusual, however, is the fact that this subroutine is called arbitrarily (the usual option was omitted). The description is brief; so it is just dumped on you without asking whether you wanted it.

Another free-standing task is called during initialization to set up the short table of binary numbers. This is strictly a reference table, so it is appropriate to generate it during startup housekeeping. It too is done somewhat arbitrarily—the table is only needed to support the computer—player mechanisms. If, as the program is exercised, the computer is never allowed to play its supporting routines are not needed. Really, you ought to let it play at least one game with you; otherwise, all of that extra coding is just taking up space.

Two tasks are associated with setting up for a given game. One is the dialog involved in getting your decisions regarding who will play. The program is the arbitrator of who will play first, second, and third. Once it completes its capricious selections, the next player subroutine is used to advise you of the playing order.

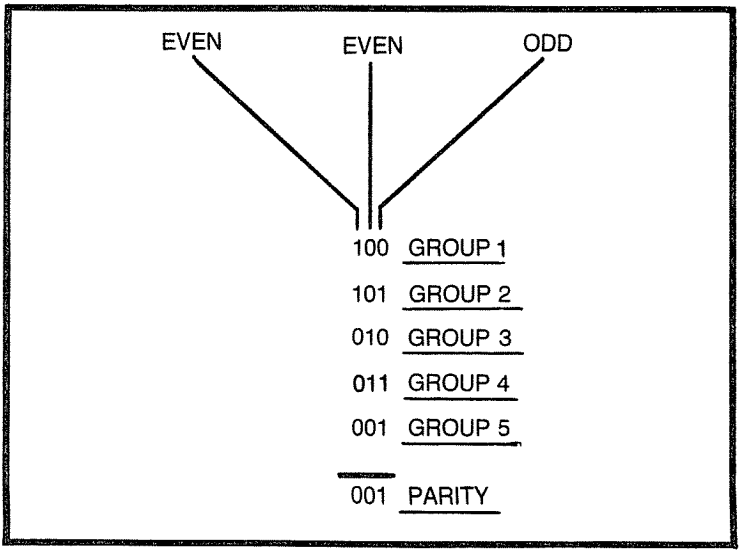


Fig. U-2. The parity of Ultranim tokens overall.

From that point on the mainline is in control. The next player is announced, a game line is printed, whoever is up is allowed to play, and a winning situation is tested for. If the game is not yet over, the mainline cycles through the whole sequence again. The really interesting things to look for in the coding is how the program knows who is up and that business of the computer's play.

Complying with the doctrine of most programming schools, all of the required tables are declared up front. Line 50 sets aside the binary number table. It's called *B*, and it provides for three binary integers per line, for the range of zero through five. *Table A*, which is another array, is also declared in line 50; the *A* is for analyzer. More about that later.

The final act of setting up for a game is a presentation of the order of play. This is economically done by the series of GOSUB statements in lines 3280 through 3300. Those jumps are all to line 3400, which is the start of the next player module. This is the same subroutine that is called preparatory to each player's turn. The way that routine works is rather simple.

Table U, which has the scrambled series of 1-2-3, is operated as a circulating stack. The topmost element, having either a one, a two, or a three in it, is used as a subscript to print the corresponding *U$* constant—but only if that constant was not blanked out. In preparation for the next time this module will be called, then, the numbers in the *U* stack are bumped up one line, with the used number being placed on the bottom of the stack. This all works to maintain the order of play and to print out the appropriate player prompts with a single, simple coding sequence.

The next thing that the players see after their prompt is the game line, after its having been updated according to the previous player's move. The print commands in lines 4000 through 4040 output the *ribbon line*, and a pair of FOR-NEXT loops are used to print the tokens themselves. A TAB value is maintained in *T* so that, as the inside loop completes each group, the outside loop aligns the printing within the next group. For each number within a group, an *X* is printed—that number being the one held in the group fields of *table G*. The null print in line 4140 is to space the output up one line, just before the RETURN in line 4150.

As the game goes on all human inputs are processed by a common module. Line 5000 asks for a group number, a comma, and the number to be deleted from that group. The entry string goes into *G* and *X*.

The choice of *U*, as used in line 60, is for *Ultranim's* users. The three-field alphanumeric table (*U$*) will later contain the three

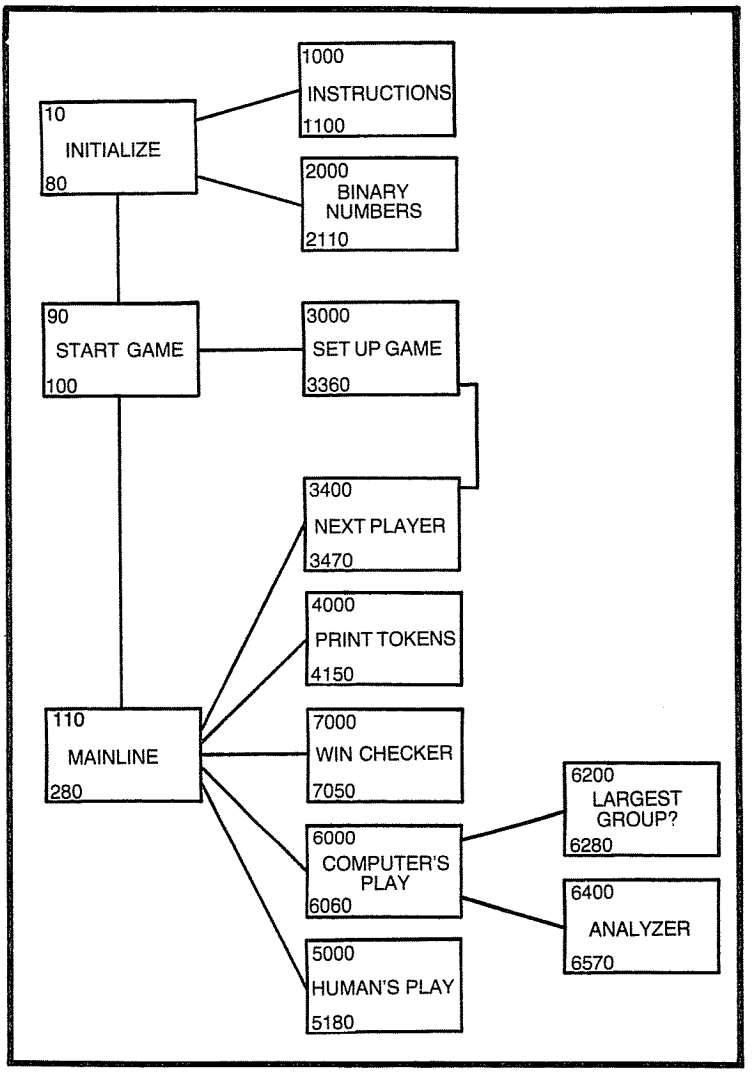


Fig. U-3. Program template for Ultranim.

prompts, YOU, FRIEND, and ME. They each also have a numeric worth (1, 2, and 3), which is what the numeric table (*U*) is used for. In line 70 the symbol *G* means group; *P* means parity. There are five groups, so *G* needs five fields. To do the vertical parity scheme on the number of tokens in all of the groups requires three columnar positions; *P* is given one element for each column's parity.

After establishing the table areas a jump is made out of line 80 to load up *table B* with the binary digits. The digits are grouped as

sets of three (one set per decimal equivalent) in the DATA statements of lines 2000 through 2050. A pair of concentric FOR-NEXT loops do the loading task by using a pair of subscripts with a READ, directly into *table B*. When done the RETURN in line 2110 goes back to line 90.

The GOSUB 3000 is to set up the game. The first task done there is to load the number *5* in each of the fields in *table G*. A simple FOR-NEXT loop does this (lines 3000 through 3020).

A menu type of display is shown by lines 3030 through 3060, from which the operator is supposed to indicate the player-mode choice. The series of statements from line 3070 to line 3210 are all executed while the operator is deciding. This series first loads the *U$* constants; then it does a simple unsort of the series 1-2-3 in *table U*.

The operator's decision is accepted into *U* (the simple variable) and, depending on the response (ME, FRIEND, or neither) may be blanked out in *table U$*.

Lines 5040 through 5070 insure that only positive integers of five or less are input, and lines 5080 and 5090 insure that the player's choices are logically valid. If they are line 5100 does a simple subtraction directly into *table G*. If a clumsy keystroke did result in unusable input the player is admonished by either line 5140 or line 5170, and is permitted to type in his or her choice all over.

Just before the call to get a player's move (either a human's or the computer's), the jump to line 7000 is executed out of line 130 in the mainline. A win trigger called *W* is incremented by line 7030 for every group that is not zero. As long as *W* ends up with more than one the game goes on. When *W* does come back to line 140 as a one the game might as well be ended. It is reasonable to assume the next player will grab whatever is left in that final group and win the game by reason of picking last.

"WHEE . . . " is the only message indicating the winner—the prompt has already been output, the game line is there, and it is obvious to all who is the victor. Whoever it is, he or she must have played a perfect game. The following "programmed intelligence" shows how the computer can play a perfect game.

The series of statements from line 6200 to line 6280 are all for determining which group has the most tokens in it. A match maximum worker called *M* is used by a FOR-NEXT loop (lines 6210 through 6240) to learn which *G* field is the largest. Another FOR-NEXT loop follows (lines 6250 through 6270) to load into simple variable *G* the number of that group. Two things are established then: *G* knows the group number and *M* knows how many are in that group.

The first task of the analyzer is to convert the numbers in the groups to binary. A pair of concentric FOR-NEXT loops does this, and results are fed directly into array *A*. The compound expression (of the subscripts) in line 6420 looks more confusing than it really is. The leftmost subscript for *B* uses G(I); *G* is the group table and *I* is incremented through all five groups. *J* is the column counter for the three vertical columns of binary digits in *table B*.

The next pair of nested loops does two things. The vertical columns of the array (*A*) are summed in a conventional decimal manner into the elements of *table P*. The results are individually checked by the series from lines 6500 to 6520 to determine odds and evens. The *P* elements are then rewritten with either a one or a zero accordingly.

Back up in line 6050, which is the return point from the analyzer, a simple arithmetic check is made to see whether the three *P* fields add up to nothing. If that is the case the computer has done its thing. If the parity across the board is not yet even line 6010 is called upon again to remove another token from the largest group, and the whole analysis is done again.

These reiterative processes are laborious, but they are done quite quickly on most any computer—even on the relatively slow microcomputer. If you don't yet have a full appreciation for your own micro's speed, try computing your own moves in exactly this same fashion. You too can play a perfect game of *Ultranim*—if you don't make any mistakes.

THE PROGRAM

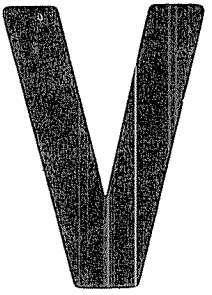
```
 10 REM  "ULTRANIM"
 20 REM
 30 GOSUB 9000
 40 GOSUB 1000
 50 DIM B(6,3), A(5,3)
 60 DIM U$(3), U(3)
 70 DIM G(5), P(3)
 80 GOSUB 2000
 90 GOSUB 3000
100 PRINT
110 GOSUB 3400
120 GOSUB 4000
130 GOSUB 7000
140 IF W = 1 THEN 200
150 IF U$ <> "ME" THEN 180
160 GOSUB 6000
```

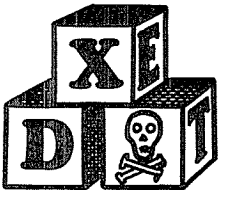
```
170 GOTO 110
180 GOSUB 5000
190 GOTO 110
200 PRINT "WHEE ..."
210 PRINT
220 PRINT "ANOTHER GAME (Y OR N)";
230 INPUT Q$
240 IF Q$ = "Y" THEN 90
250 IF Q$ = "N" THEN 270
260 GOTO 220
270 PRINT "THEN END.  ULTRANIM."
280 END
1000 PRINT "A GAME BEGINS WITH 5 GROUPS"
1010 PRINT "OF 5 TOKENS."
1020 PRINT
1030 PRINT "IN A TURN A PLAYER MAY PICK"
1040 PRINT "FROM ONE TO ALL OF THE TOKENS"
1050 PRINT "IN ANY GROUP."
1060 PRINT
1070 PRINT "THE PLAYER THAT GETS THE LAST"
1080 PRINT "PICK WINS THE GAME."
1090 PRINT
1100 RETURN
1999 REM  "BINARY NUMBERS"
2000 DATA 0, 0, 0
2010 DATA 0, 0, 1
2020 DATA 0, 1, 0
2030 DATA 0, 1, 1
2040 DATA 1, 0, 0
2050 DATA 1, 0, 1
2060 FOR I = 1 TO 6
2070 FOR J = 1 TO 3
2080 READ B(I,J)
2090 NEXT J
2100 NEXT I
2110 RETURN
2999 REM  "SET UP GAME"
3000 FOR I = 1 TO 5
3010 LET G(I) = 5
3020 NEXT I
3030 PRINT "HOW MANY PLAYERS?"
3040 PRINT "  1 = YOU AND I"
3050 PRINT "  2 = YOU AND A FRIEND"
```

```
3060 PRINT "  3 = YOU, A FRIEND, AND I"
3070 LET U$(1) = "YOU"
3080 LET U$(2) = "FRIEND"
3090 LET U$(3) = "ME"
3100 LET U(1) = 1
3110 LET U(2) = 2
3120 LET U(3) = 3
3130 FOR I = 1 TO 3
3140 LET T = INT(10*RND(1))+1
3150 IF T > 3 THEN 3140
3160 LET U = U(I)
3170 LET U(I) = U(T)
3180 LET U(T) = U
3190 NEXT I
3200 PRINT "#";
3210 INPUT U
3220 IF U = 1 THEN 3350
3230 IF U = 2 THEN 3330
3240 IF U = 3 THEN 3280
3250 PRINT "NIX!  ANSWER WITH 1, 2, OR 3."
3260 GOTO 3200
3270 PRINT "THE ORDER OF PLAY IS:"
3280 GOSUB 3400
3290 GOSUB 3400
3300 GOSUB 3400
3310 PRINT
3320 RETURN
3330 LET U$(3) = " "
3340 GOTO 3290
3350 LET U$(2) = " "
3360 GOTO 3290
3399 REM  "NEXT PLAYER"
3400 LET U = U(1)
3410 LET U$ = U$(U)
3420 LET U(1) = U(2)
3430 LET U(2) = U(3)
3440 LET U(3) = U
3450 IF U$ = " " THEN 3400
3460 PRINT U$
3470 RETURN
3999 REM  "PRINT TOKENS"
4000 PRINT " ..1..";
4010 PRINT " ..2..";
```

```
4020 PRINT " ..3..";
4030 PRINT " ..4..";
4040 PRINT " ..5.."
4050 LET T = 1
4060 FOR I = 1 TO 5
4070 PRINT TAB(T);
4080 FOR J = 1 TO G(I)
4090 IF G(I) = 0 THEN 4120
4100 PRINT "X";
4110 NEXT J
4120 LET T = T+6
4130 NEXT I
4140 PRINT
4150 RETURN
4999 REM  "HUMAN INPUT"
5000 PRINT "(G,X)";
5010 INPUT G, X
5020 LET G = INT(ABS(G))
5030 LET X = INT(ABS(X))
5040 IF G < 1 THEN 5120
5050 IF X < 1 THEN 5120
5060 IF G > 5 THEN 5120
5070 IF X > 5 THEN 5120
5080 IF G(G) < 1 THEN 5140
5090 IF X > G(G) THEN 5170
5100 LET G(G) = G(G)-X
5110 RETURN
5120 PRINT "I DON'T UNDERSTAND."
5130 GOTO 5000
5140 PRINT "GROUP"G"IS EMPTY";
5150 PRINT " -- TRY AGAIN."
5160 GOTO 5000
5170 PRINT "GROUP"G"HAS ONLY"G(G);
5180 GOTO 5150
5999 REM  "COMPUTER'S PLAY"
6000 GOSUB 6200
6010 LET G(G) = G(G)-1
6020 IF G(G) <> 0 THEN 6040
6030 RETURN
6040 GOSUB 6400
6050 IF P(1)+P(2)+P(3) <> 0 THEN 6010
6060 RETURN
6199 REM  "LARGEST GROUP"
```

```
6200 LET M = G(1)
6210 FOR I = 2 TO 5
6220 IF M > G(I) THEN 6240
6230 LET M = G(I)
6240 NEXT I
6250 FOR G = 1 TO 5
6260 IF G(G) = M THEN 6280
6270 NEXT G
6280 RETURN
6399 REM  "ANALYZER"
6400 FOR I = 1 TO 5
6410 FOR J = 1 TO 3
6420 LET A(I,J) = B(G(I),J)
6430 NEXT J
6440 NEXT I
6450 FOR I = 1 TO 3
6460 LET P(I) = 0
6470 FOR J = 1 TO 5
6480 LET P(I) = P(I)+A(J,I)
6490 NEXT J
6500 IF P(I) = 0 THEN 6540
6510 LET T = INT(P(I)/2)
6520 IF T*2 = P(I) THEN 6560
6530 LET P(I) = 1
6540 NEXT I
6550 RETURN
6560 LET P(I) = 0
6570 GOTO 6540
6999 REM  "WIN CHECKER"
7000 LET W = 0
7010 FOR I = 1 TO 5
7020 IF G(I) = 0 THEN 7040
7030 LET W = W+1
7040 NEXT I
7050 RETURN
9000 REM  "RANDOM NUMBER ROUTINE"
9010 LET Z = RND(1)
9020 RETURN
```





Verboten

Imagine a cosmopolitan computer. Here is a chance to teach your electronic companion a bit of German. Although other languages could be equally possible the German equivalent for forbidden quite nicely fits our chapter-per-letter scheme.

This game is a derivative of the basic theme of players guessing a hidden value that was chosen at random by the computer. Its construction is simple, yet the game has quite a lot more appeal than the variety that has no more substance than to just pick a number.

Whether played by German kinder or children elsewhere, they all enjoy the suspense that builds as they take turns picking letters of the alphabet. Eventually, in twenty-six or fewer turns, the alphabet will run out.

Three things can happen with each turn. The player may get to keep his or her chosen letter and score one more point. Some one letter, when picked, will declare a bonus and that player's score for the round thus far will be doubled. Some other letter is *verboten*. The player who gets the forbidden one loses his or her score instantly. Another round is started then, and the game goes on.

Presumably no lengthy argument is necessary to convince you to adopt this program into your games library. The few minutes required to copy it could hardly be an excuse not to, and there is at least one programming nicety within it. And, it is fun to play.

A VERBOTEN PROGRAM

A quick glance at Fig. V-1 shows nearly all of the structures in this program. As implied by the drawing a primary task during

program initialization is to read the alphabet as DATA characters, and to store them in a table. The table is used thereafter for obtaining numerical equivalents of the letters that are typed in by the players.

Three short tasks are involved in preparing for a round of play. There is a *tank* that is used to know which letters are owned at any point by either of the two players. The tank, which is really just a numeric table, has to be cleared before each new round. The other two housekeeping chores, per round, is the picking of the bonus letter and the one that is forbidden. As can be surmised, an RND function is used in both cases, and that is why the player entries must be reduced to numerical substitutes. (The letter codes are

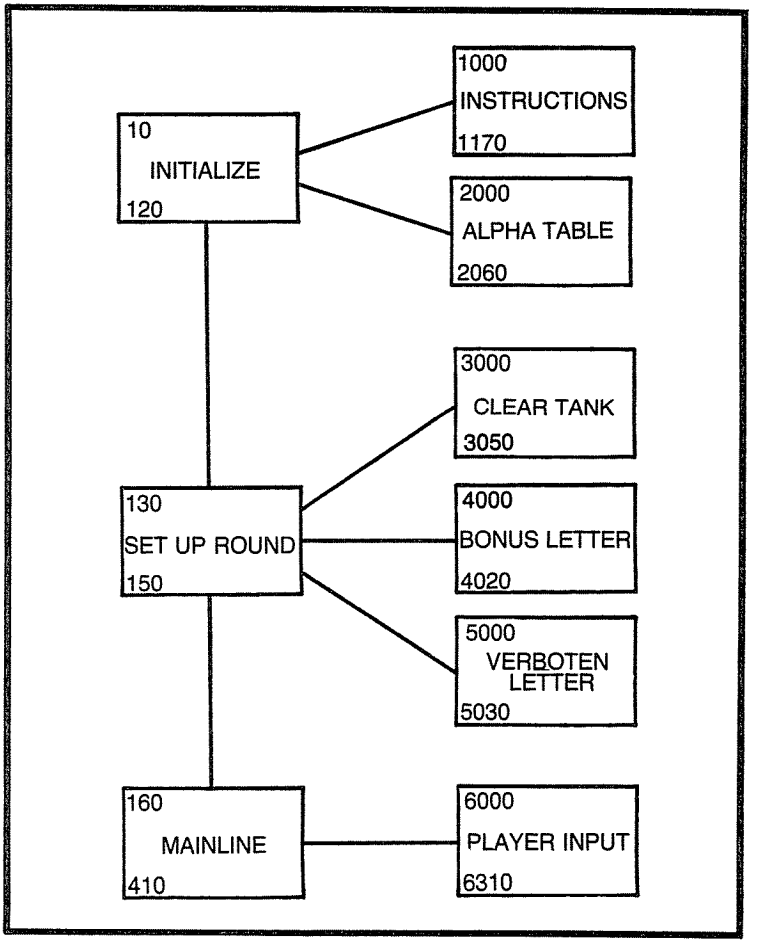


Fig. V-1. Program template for Verboten.

repeatedly compared to the two random integers that are being held in suspense.)

Most of the program's mainline has to do with scorekeeping and the end-of-game dialog. There is but one jump from the mainline to the input module. It is there that all of the tasks associated with player entries are done. Player alternation is also done in the input module— as they acquire their letters, the tank is posted with their code number.

It is the use of the tank and the codes therein that gives this program a distinct character. Often, to impart a sense of personalized dialog to a game the players are asked to input their names. In *Verboten* a technique is used that instead gives an implied awareness as to which player is whom. The awareness is derived from the codes in the tank.

Depending on which of the two players is at the keyboard the computer uses the pronouns *you* and *your*—and it can tell them apart! If you are up and you type a letter that you picked before the program tells *you* that *you* already own that letter. On the other hand, if you type a letter that is owned by the other player, the message says that the letter is owned by *your* friend.

This can be a refreshing technique occasionally. Personalized games are fun, but it can be monotonous having to announce yourself game after game. Looking through the listing it can be seen how easy it is to plug in this nicety and how easy it can be to add that cosmopolitan flair as well.

VERBOTEN'S CODING

The DIM statement in line 40 has all of the declaratives. The table called *A$* is the one for the alphabet. The one called *X* is the tank. To be able to use the player number as a subscript for round totals *table R* is declared: one element for each player. Later, the simple form of *R* is also used, again meaning round, but in this case the total there is the cumulative rounds count.

The remainder of the program's initialization is typical enough, ending with the jump out of line 120 to generate the alphabet's table (lines 2000 through 2060). A 26-times FOR-NEXT loop is used there to load the alpha constants into *table A$*.

Lines 130, 140, and 150 comprise the start of a round. The three GOSUB statements are executed in quick succession, one to each of the separately identified housekeeping tasks. Obviously, all three tasks could have been included in a single module. My predilection for task-level modularity prevails, however— this does make it easy to describe the tasks one at a time.

To clear the tank a simple FOR—NEXT loop is used (lines 3020 through 3040) to overwrite 0s into all twenty-six locations of *table X*. In lines 3000 and 3010, the players' round totals are cleared also, with two simple move expressions.

The picking of a bonus letter is pure simplicity. The expression in line 4000 will place into *B* (for bonus) an integer in the range of one to one hundred. Only those numbers one to twenty-six are usable; so line 4010 will refuse anything larger and will allow the RETURN when RND finally gives a valid number.

Setting up the *V* worker (verboten) is done the same way. Another statement was added in this case. (line 5020) to preclude the same letter coming up as the bonus and the forbidden one. This is done by simply comparing *B* to *V*, and if equal, insisting on another call to RND. With both workers loaded with different integers the return in line 5030 will let the round get under way.

The jump from the mainline to the input module is immediate out of line 160, and it will be done repeatedly until *X$* is returned empty. The conditional in line 170 maintains this tight loop until *X$* signals that a round of play has been concluded. The variable *X$* is the one that is used for player entries. To see it in use, we look at the remaining major module.

The first four lines of the player input module (lines 6000 through 6030) take care of the variable *P*. This logic is analogous to a flip-flop. By the time line 6040 is reached which outputs the input prompt, the player number that is in *P* will be either a one or a two. Whichever is there will automatically flip-flop to the other any time this module is entered from the top.

The player is being asked at this point to enter a letter. The entry will be accepted into *X$* at line 6060. To learn whether the players type anything useful, the FOR-NEXT sequence from line 6070 to line 6090 does a quick scan to compare *X$* with the letters of the alphabet (that is in *table A$)*. Any match will cause a branch to line 6130—at least a letter was typed in, and it was not cluttered or surrounded by any extraneous keystrokes.

If it should happen that the loop completes and falls through without finding a match, *X$* has unintelligible junk in it. Loosely translated, the expression FRISCH AUF! means to *look alive*. This error message is followed by a branch back to the prompt line to permit another entry by the same player.

Where a good letter was found in *X$*, at that point *I* has its relative equivalent number. Line 6130 uses that number to look into the tank to see whether this same letter has ever been typed before.

If *table X* has a zero in the *I* location, we have in fact a new letter: go on through, otherwise go to line 6260 to handle the error.

Assuming for the moment that *X$* does contain a single letter that has not been tried previously it is time to see whether it matches either the bonus or the verboten one, and to process it accordingly. If it is the verboten one, as detected by the comparison in line 6150, the branch is to line 6220 where "VERBOTEN!" is announced. The player's scorekeeper (for the round) is zeroed, *X$* is blanked, and the return to the mainline is out of line 6250.

The bonus play is detected by line 6160, which compares the *I* counter to the RND value being held in *B*: if they are equal, "*** BONUS***" is announced. Line 6200 doubled the player's round score, and the return is out of line 6210. In this case *X$* is not blanked. Back in the mainline if any residue does remain in *X$* the round goes on.

If the tests for *V* and *B* both fail line 6170 simply adds one point to *R(P)*. Again, *X$* is allowed to retain its content because the play was valid and a return is taken out of line 6180.

Now for the duplicates type of error. When the tank is looked at by line 6130, and it does not find a zero, what it does find is either a one or a two. One of these two codes got there by reason of the statement in line 6140, through which all valid plays pass. In line 6260 then it is the code number (the player number) that is being examined specifically.

It is easily known whether the previously played letter was picked by you or your friend. Your number is currently in *P*. Whichever number you are, the other number must belong to your friend. After the message is printed, the branch is back to line 6050 to let you (or your friend) try again.

Sooner or later one of you will pick the verboten letter. In line 170, in the mainline, when *X$* finally does come back with a blank in it the round is ended. The round scores are added to *T1* and *T2*, which are your respective totals. You can either continue or stop. If you do want to stop now it's AUF WIEDERSEHEN.

THE PROGRAM

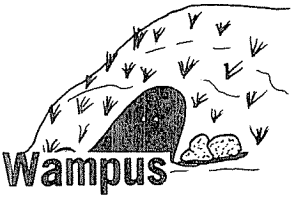
```
10 REM  "VERBOTEN"
20 REM
30 GOSUB 9000
40 DIM A$(26), X(26), R(2)
50 PRINT "WANT INSTRUCTIONS (Y OR N)";
60 INPUT Q$
70 IF Q$ = "Y" THEN 110
```

```
 80 IF Q$ = "N" THEN 120
 90 PRINT "WAS GIBT ES?"
100 GOTO 50
110 GOSUB 1000
120 GOSUB 2000
130 GOSUB 3000
140 GOSUB 4000
150 GOSUB 5000
160 GOSUB 6000
170 IF X$ <> " " THEN 160
180 LET T1 = T1+R(1)
190 LET T2 = T2+R(2)
200 LET R = R+1
210 PRINT "ROUND #"R"     TOTAL"
220 PRINT "# 1 ="R(1)"       "T1
230 PRINT "# 2 ="R(2)"       "T2
240 PRINT
250 PRINT "GO AGAIN (Y OR N)";
260 LET Q$ = " "
270 INPUT Q$
280 IF Q$ = "Y" THEN 130
290 IF Q$ = "N" THEN 320
300 PRINT "WAS ES LOS?"
310 GOTO 250
320 PRINT "NEW GAME (Y OR N)";
330 LET Q$ = " "
335 INPUT Q$
340 IF Q$ = "N" THEN 400
350 LET T1 = 0
360 LET T2 = 0
370 IF Q$ = "Y" THEN 130
380 PRINT "WAS GIBT ES?"
390 GOTO 320
400 PRINT "AUF WIEDERSEHEN"
410 END
1000 PRINT "YOU AND A FRIEND TAKE TURNS."
1010 PRINT "TYPE ANY LETTER OF THE ALPHABET"
1020 PRINT "THAT HASN'T ALREADY BEEN PICKED."
1030 PRINT "EACH LETTER IS WORTH ONE POINT,"
1040 PRINT "EXCEPT:  (IN A ROUND)"
1050 PRINT "  ONE IS A BONUS, WHICH WILL"
1060 PRINT "  DOUBLE YOUR SCORE, AND, ONE"
1070 PRINT "  IS VERBOTEN, WHICH WILL ERASE"
```

```
1080 PRINT "  YOUR SCORE."
1090 PRINT "WHEN THE FORBIDDEN LETTER IS"
1100 PRINT "PICKED THE ROUND ENDS AND YOUR"
1110 PRINT "SCORES ARE UPDATED."
1120 PRINT "THE NEXT ROUND BEGINS WITH A"
1130 PRINT "NEW ALPHABET."
1140 PRINT "                    READY";
1150 INPUT Q$
1160 PRINT
1170 RETURN
1999 REM  "ALPHA TABLE"
2000 DATA A,B,C,D,E,F,G,H,I,J,K,L,M
2010 DATA N,O,P,Q,R,S,T,U,V,W,X,Y,Z
2020 FOR I = 1 TO 26
2030 READ A$(I)
2040 NEXT I
2050 RESTORE
2060 RETURN
2999 REM  "CLEAR TANK"
3000 LET R(1) = 0
3010 LET R(2) = 0
3020 FOR I = 1 TO 26
3030 LET X(I) = 0
3040 NEXT I
3050 RETURN
3999 REM  "BONUS LETTER"
4000 LET B = INT(100*RND(1))+1
4010 IF B > 26 THEN 4000
4020 RETURN
4999 REM  "VERBOTEN LETTER"
5000 LET V = INT(100*RND(1))+1
5010 IF V > 26 THEN 5000
5020 IF V = B THEN 5000
5030 RETURN
5999 REM  "PLAYER INPUT"
6000 IF P <> 1 THEN 6030
6010 LET P = 2
6020 GOTO 6040
6030 LET P = 1
6040 PRINT "PLAYER #"P;
6050 LET X$ = " "
6060 INPUT X$
6070 FOR I = 1 TO 26
```

```
6080 IF X$ = A$(I) THEN 6130
6090 NEXT I
6100 PRINT "FRISCH AUF!"
6110 PRINT
6120 GOTO 6040
6130 IF X(I) <> 0 THEN 6260
6140 LET X(I) = P
6150 IF I = V THEN 6220
6160 IF I = B THEN 6190
6170 LET R(P) = R(P)+1
6180 RETURN
6190 PRINT "*** BONUS ***"
6200 LET R(P) = R(P)*2
6210 RETURN
6220 PRINT "VERBOTEN!"
6230 LET R(P) = 0
6240 LET X$ = " "
6250 RETURN
6260 IF X(I) = P THEN 6300
6270 PRINT "YOUR FRIEND ";
6280 PRINT "PICKED "X$" BEFORE"
6290 GOTO 6050
6300 PRINT "YOU ";
6310 GOTO 6280
9000 REM  "RANDOM NUMBER ROUTINE
9010 LET Z = RND(1)
9020 RETURN
```

W



Have you ever seen a *wampus?* "He's a big hairy creature that will gobble you if he gets you." So goes the story that introduces this game.

The wampus lives in a cave. You are lost in his cave trying to find the only exit so that you can go home. There are twenty chambers in the cave and each has three passageways leading to other chambers. One of the tunnels does go to the cave' mouth, and when you find it you are free.

There are hazards. In one chamber there is a deep pit. If you stumble into there you have had it. In three other chambers there are bats. You are afraid of bats, and when you encounter them you take the nearest exit. Normally you have time to ponder over which of the three tunnels to take next.

When you wander into the wampus you have another choice. He is afraid of loud noises. You have a six-shooter. If you crawl into the chamber where he is you can make a hasty exit; or if you fire the gun he will take off to another chamber. You are safe for the moment. You don't know which tunnel he took, but neither does he know where you will go next. If ever you run out of ammunition and and happen to run into the wampus again, he will eat you.

DEFINING THE PROBLEM

Looking through the game's description there are the following parameters to contend with.

- There is one exit tunnel
- There is one pit (one of the chambers)
- Three chambers have bats in them
- The gun has six shots
- There are twenty chambers
- Each chamber has three tunnels going from it

The matter of the cave itself should be dealt with first. To conceive of a floor plan that would have twenty rooms, all interconnected such that each has three passages going to others, implies the need for a definite geometric pattern. One such possibility is a decagon (a ten-sided polygon). As shown in Fig. W-1 there are actually two decagons used in *Wampus*, one inside of the other.

Each of the cave rooms is assigned a number from the series one to twenty. The numbering scheme is arbitrary and it remains constant. All of the other parameters vary from game to game. As each game is set up, one room is randomly designated the exit; another has the pit, and three others have bats in them.

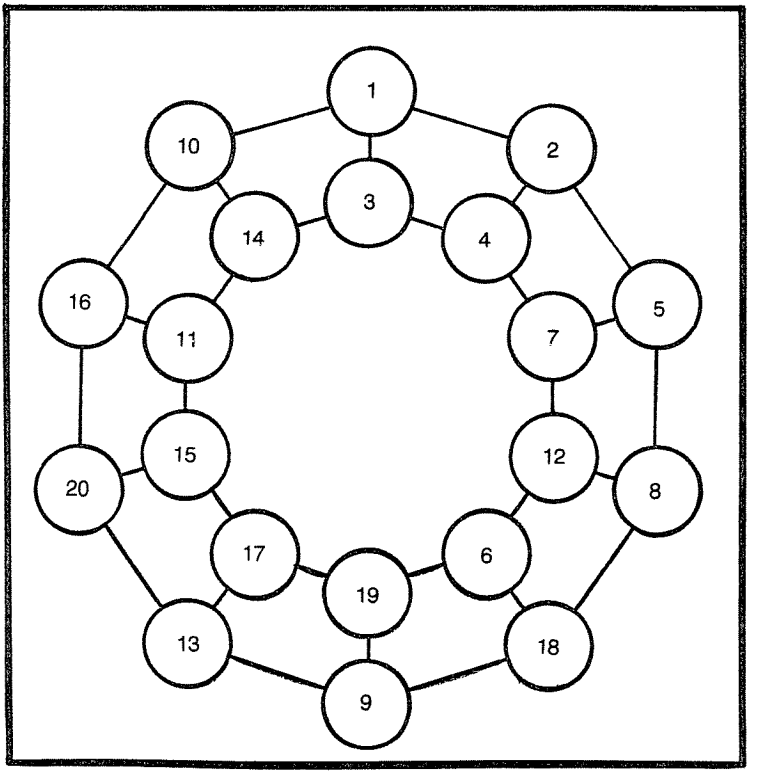


Fig. W-1. A floor plan of the cave where the man-eating wampus lives.

During play you and the wampus move from room to room. The ability to do so depends on knowing which room numbers can be reached from any one room.

DESCRIBING THE PROGRAM

As the program is being initialized the player is afforded an option to receive a printout of the story. This capability is alluded to in the program template for *Wampus,* as shown in Fig. W-2. The block numbers 1000 through 1230 imply that this is more of a story than merely instructions. It is.

To set up the game is really to set up the cave. A free-standing subroutine is used to get room numbers at random, and the numbers are then allocated to the exit, the pit, the bats, you, and wampus. Control from there on is maintained by the mainline.

As you run through the cave, and as you enter each chamber, you must pick the number of the tunnel to go through next. This is so when you have time, anyway. When you run into bats the program will help you by making the exit choice for you. Normally the dialog goes thus:

```
YOU ARE IN ROOM 2
EXITS ARE: 1 4 5
MOVE TO?
```

Take your pick and enter either 1, 4, or 5. Your room number changes according to your choice, and the dialog is repeated. If you run into bats the message is: "THERE ARE BATS IN HERE, SO YOU RUN TO:"—the message is followed by your new room number, automatically.

If you bump into the wampus another subroutine is called. You are first offered the choice: SHOOT OR RUN (S OR R). If you answer that you want to run the dialog from above comes back into play. If you decide to shoot: BANG... HE RAN OUT. Control then returns to the normal-move sequence.

If ever YOU CAN FEEL A DRAFT, one of the three exits available to you leads to the mouth of the cave. Remember the other two numbers if you do not pick the winner. Taking a wrong turn does eliminate one of the numbers just shown to you.

Go ahead, but watch out for the wampus. Go easy on the use of the gun, too. After you have fired six times, and you run into him again: "YOU'RE OUT OF SHOTS, AND WAMPUS HAS YOU."

DESCRIBING THE MECHANICS

There is a series of DATA statements inside the program that contain sixty numbers. That is, there are three sets of twenty

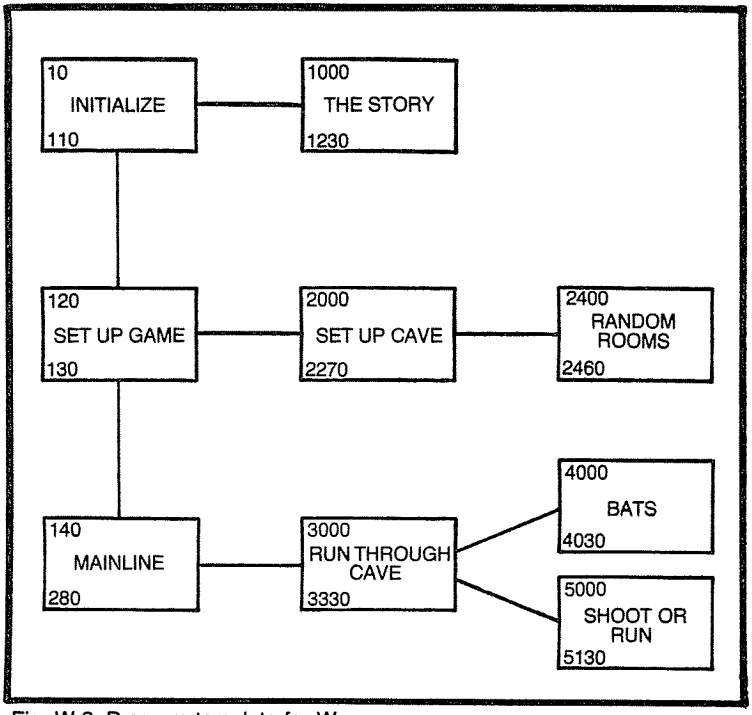


Fig. W-2. Program template for Wampus.

numbers. They are accessed and used, both during the setup and the play, by a belt READ loop.

From line 130 in the program's listing there is a GOSUB 2000. It is in lines 2010 through 2040 that the numbers are established as DATA constants. The jump from line 2050 goes directly to the random rooms module—that is, to the belt READ loop.

In line 2400 the variable *R* is loaded with a random value of one to one hundred. The belt loops then until *R* is reached. The READ statement in line 2420 is inside of the loop, and each time it is executed three numbers are obtained from the DATA string. There is also, immediately after the READ, a conditionally executed RESTORE. The only time that *R2* ever comes up with a sixteen is at the end of the list. By restoring the DATA pointer at that point the DATA belt can continue uninterrupted. (The list of numbers is constructed intentionally so that the last set of three has a unique *R2* value; it could be any number so long as it doesn't occur elsewhere in the same relative position.

During setup the belt READ loop is called four times. The first time (out of line 2050) is to get the exit number. Variables *R1, R2,*

and *R3* are returned with valid interconnecting numbers. Somewhat arbitrarily, *R3* is loaded into *E* as the cave exit number. One of the bat caverns is picked at this point also. The number that is in *R1* is placed into *B1* in line 2070.

More numbers are needed. Another GOSUB 2400 is done out of line 2050. This time the *P* (for pit) variable is loaded with *R1*, and a second bat room (*B2*) is taken from *R3*. Some exclusives are necessary at this point.

Line 2110 makes sure that the pit and the exit are not the same. Lines 2120 and 2130 insure that the bats are not in the pit chamber either. If any of these tests are true, or if both *B2* and *B3* are the same number, this phase is executed again by going back to line 2080.

The remaining bat chamber, *B3*, is obtained by another jump to line 2400. Another series of tests is done in lines 2170 through 2200. These work the same as those previous, this time making sure that *B3*. which was loaded from *R2*, is not contrary to the exclusives.

You and the wampus are next. The fourth jump to line 2400 is made out of line 2210. The variable *Y* is for *you*; you start out in room *R3*. The wampus is known as *W*; he begins the game in *R1*. The *X* variable that is turned on with a one in line 2260 will be used later to let the mainline know when the game is over. If *X* is ever passed to the mainline with a zero in it, the end-of-game dialog is invoked.

Primary mechanics as you run through the cave are done by lines 3000 through 3330. From the top, *Y*, which is where you are, is tested for several conditions. If your room number matches that of the exit (*E*), the branch that is taken is through line 3230, followed by a return to the mainline. Because the *X* trigger is turned off in line 3230 the mainline will know the game is over.

Line 3010 tests whether you have fallen into the pit. If *Y* ever does match *P* the same exit is taken back to the mainline. The end-of-game dialog will discern why *X* was turned off.

Another test that is made, presuming the game can go on, is whether you have just entered a room where the wampus is. If your room number is the same as *W*, the branch is to line 3270. If you are out of shots the usual exit is taken back to the mainline; otherwise, the module that runs from line 5000 to 5130 is called. More about that in a moment—there are more tests in lines 3030 through 3050.

The bats are in rooms *B1*, *B2*, and *B3*. Whenever *Y* comes up the same as any *B* field the branch is to line 3250. That sequence first calls line 4000, then branches to 3000. All of that happens in the bats module is that *Y* is reloaded with whatever is in *R1;* that is the nearest exit. After you are told your new room number all of the

preceding tests are again executed. Your hasty exit may have landed you in trouble.

If all of that series of tests fail (lines 3000 through 3050) a normal move is in order. Line 3060 tells you what room you are in. Line 3070 does a RESTORE to reset the DATA belt to its starting point. Another version of the READ loop is used then, to load into *R1*, *R2*, and *R3* the exit possibilities from the room you are in. This is done by running down the belt until it is aligned with *Y*. That is where you are.

The possible exit paths are shown to you by line 3110. Before the prompt that asks for your choice is reached (in line 3160) the program checks whether any of them are the way out. If so you are alerted to the possibility by the message that is printed by line 3150. Being able to feel a draft is supposed to mean that you are near the cave entrance.

It is interesting to notice that, by reason of the testing sequences, whenever you are in a room where there are bats you leave so quickly that you do not have time to feel any draft—even if there was one.

The bit of logic that was passed by a moment ago is that having to do with encounters with our hairy friend. Out of line 5010 is where you have the option to shoot or run. If you respond with the letter *R* the program simply goes to the normal move sequence. If you shoot the gun by typing an *S* the wampus will do the running.

Normally the wampus will run through the middle tunnel *(R2)*, as can be seen in line 5100. An exception is that he will not leave the cave. If *R2* is the same as *E* (the exit), the wampus will instead go to *R1*. It is possible that he will go into a cave where there are bats. The tests are all ordered such that the wampus business takes precedence over that having to do with the bats. After all, he represents a more serious encounter than the bats.

It is also possible that the wampus will run into the chamber that has the deep pit. That is all right for him—he can crawl the walls. If you run into that same room you are done for. The earlier test sequence traps out if *Y* equals *P*—before the test for whether *Y* equals *W*. Either way you are done for. So are we.

THE PROGRAM

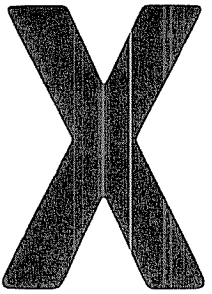
```
10 REM  "WAMPUS"
20 REM
30 GOSUB 9000
40 PRINT
50 PRINT "WANT THE STORY (Y OR N)";
```

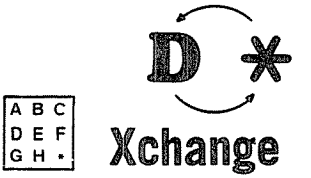
```
 60 INPUT Q$
 70 IF Q$ = "Y" THEN 110
 80 IF Q$ = "N" THEN 120
 90 PRINT "DON'T YOU KNOW?"
100 GOTO 40
110 GOSUB 1000
120 PRINT
130 GOSUB 2000
140 GOSUB 3000
150 IF X <> 0 THEN 140
160 IF Y <> E THEN 200
170 PRINT "YOU GOT OUT, BUT WAMPUS"
180 PRINT "WILL GET YOU NEXT TIME."
190 GOTO 220
200 IF Y <> P THEN 230
210 PRINT "YOU FELL INTO THE PIT;"
220 PRINT "SO LONG ..."
230 PRINT
240 PRINT "ANOTHER GAME (Y OR N)";
250 INPUT Q$
260 IF Q$ = "Y" THEN 120
270 PRINT "BYE"
280 END
1000 PRINT "YOU ARE LOST IN A CAVE THAT HAS"
1010 PRINT "20 CHAMBERS.  ALL CHAMBERS HAVE"
1020 PRINT "TUNNELS TO THREE OTHERS.  ONE OF"
1030 PRINT "THE TUNNELS GOES TO THE MOUTH OF"
1040 PRINT "THE CAVE.  IF YOU FIND IT YOU"
1050 PRINT "GET HOME FREE."
1060 PRINT
1070 PRINT "WAMPUS LIVES IN THE CAVE.  HE'S"
1080 PRINT "A BIG HAIRY CREATURE THAT WILL"
1090 PRINT "GOBBLE YOU IF HE GETS YOU.  HE"
1100 PRINT "WILL RUN TO ANOTHER CHAMBER IF"
1110 PRINT "YOU SHOOT AT HIM.  YOUR GUN ONLY"
1120 PRINT "HAS SIX SHOTS, HOWEVER, SO USE"
1130 PRINT "THEM WISELY."
1140 PRINT "       HIT ANY KEY TO CONTINUE."
1150 INPUT Q$
1160 PRINT "SOME CHAMBERS HAVE BATS IN THEM."
1170 PRINT "THEY SCARE YOU SO BAD YOU SIMPLY"
1180 PRINT "RUN THROUGH THE NEAREST TUNNEL."
1190 PRINT "ONE CHAMBER HAS A DEEP PIT --"
```

```
1200 PRINT "DON'T FALL IN OR YOU'LL NEVER"
1210 PRINT "GET HOME."
1220 PRINT
1230 RETURN
1999 REM  "SET UP CAVE"
2000 LET S = 6
2010 DATA 2,3,10,1,4,5,1,4,14,2,3,7,2,7,8
2020 DATA 12,18,19,4,5,12,5,12,18,13,18,19,1,14,16
2030 DATA 14,15,16,6,7,8,9,17,20,3,10,11,11,17,20
2040 DATA 10,11,20,13,15,19,6,8,9,6,9,17,15,16,13
2050 GOSUB 2400
2060 LET E = R3
2070 LET B1 = R1
2080 GOSUB 2400
2090 LET P = R1
2100 LET B2 = R3
2110 IF P = E THEN 2080
2120 IF P = B1 THEN 2080
2130 IF P = B2 THEN 2080
2140 IF B1 = B2 THEN 2080
2150 GOSUB 2400
2160 LET B3 = R2
2170 IF B3 = E THEN 2150
2180 IF B3 = P THEN 2150
2190 IF B3 = B1 THEN 2150
2200 IF B3 = B2 THEN 2150
2210 GOSUB 2400
2220 LET W = R1
2230 LET Y = R3
2240 IF Y = P THEN 2210
2250 IF Y = E THEN 2210
2260 LET X = 1
2270 RETURN
2399 REM  "RANDOM ROOMS"
2400 LET R = INT(100*RND(1))+1
2410 FOR I = 1 TO R
2420 READ R1, R2, R3
2430 IF R2 <> 16 THEN 2450
2440 RESTORE
2450 NEXT I
2460 RETURN
2999 REM  "RUN THROUGH CAVE"
```

```
3000 IF Y = E THEN 3230
3010 IF Y = P THEN 3230
3020 IF Y = W THEN 3270
3030 IF Y = B1 THEN 3250
3040 IF Y = B2 THEN 3250
3050 IF Y = B3 THEN 3250
3060 PRINT "YOU ARE IN ROOM #"Y
3070 RESTORE
3080 FOR I = 1 TO Y
3090 READ R1, R2, R3
3100 NEXT I
3110 PRINT "EXITS ARE: "R1; R2; R3
3115 IF R1 = E THEN 3150
3120 IF R2 = E THEN 3150
3130 IF R3 = E THEN 3150
3140 GOTO 3160
3150 PRINT "YOU CAN FEEL A DRAFT"
3160 PRINT "MOVE TO";
3170 INPUT Y
3180 IF Y = R1 THEN 3240
3190 IF Y = R2 THEN 3240
3200 IF Y = R3 THEN 3240
3210 PRINT "TRY AGAIN."
3220 GOTO 3160
3230 LET X = 0
3240 RETURN
3250 GOSUB 4000
3260 GOTO 3000
3270 IF S > 0 THEN 3310
3280 PRINT "YOU'RE OUT OF SHOTS, AND"
3290 PRINT "WAMPUS HAS YOU."
3300 GOTO 3230
3310 GOSUB 5000
3320 IF Q$ = "R" THEN 3070
3330 GOTO 3240
3999 REM  "BATS"
4000 PRINT "THERE ARE BATS IN HERE, SO"
4010 PRINT "YOU RUN TO: "R1
4020 LET Y = R1
4030 RETURN
4999 REM  "SHOOT OR RUN"
5000 PRINT "WAMPUS IS IN THIS ROOM."
5010 PRINT "WANT TO SHOOT OR RUN (S OR R)";
```

```
5020 INPUT Q$
5030 IF Q$ = "S" THEN 5080
5040 IF Q$ = "R" THEN 5070
5050 PRINT "ANSWER QUICKLY!"
5060 GOTO 5010
5070 RETURN
5080 PRINT "BANG ... HE RAN OUT."
5090 LET S = S-1
5100 LET W = R2
5110 IF R2 <> E THEN 5130
5120 LET W = R1
5130 RETURN
9000 REM  "RANDOM NUMBER ROUTINE"
9010 LET Z = RND(1)
9020 RETURN
```





Yes, *exchange* is misspelled—intentionally. It wasn't easy to invent a game with a name that begins with *X*. It's likely though, you'll forgive the typo when you savor the flavor of what follows.

For this, our twenty-fourth *entremets*, I have concocted a program for two mortals. Or it may be enjoyed solitaire. Before a game begins you are asked whether one or two players wish to play. This program will then manage one or two *Xchange* playing grids, depending on your answer to the mode question.

THE GAME

A playing grid consists of the first eight letters of the alphabet (A through H) and an asterisk, arranged in a three by three matrix like this:

```
D H C

  B F*

A E G
```

For two players there are two of these grids side by side, and in the beginning they are identically scrambled. The method of play is to *exchange* the letters with the asterisk, one at a time, until the letters are put into their proper order with the asterisk trailing (lower right corner). There is one more rule: in a turn only a letter that is

immediately to the left or the right of, or immediately above or below the asterisk may be exchanged.

A move is the entry of just the letter; the program will then exchange the print positions of your choice and the asterisk. If you are in the dual-player mode, your opponent is permitted to make an entry also before two new grids are printed. When playing solitaire there is only the one board; so the updated output is nearly instantaneous after each entry.

In either mode, as can be imagined, the rounds of play can be very fast. Yet, if you are clumsy on the keyboard, or are not in possession of the secrets for shuffling strings, this game can drive you berserk. On the off chance that your local competition may buy his or her own copy of this book I'll not disclose here the tricky technique that will unscramble a grid in the least number of moves

There is a risk that some nonprogrammer types may stay baffled forever; so your computer could stay entrapped in the enter-print-enter cycle for all time. Rather than having to pull the plug to force an abort there is a privileged entry to permit programmers to exit to the end-of-game routine. (An X will do it in the left-hand player's position.) This fact is not advertised in the game's instruction module, however, to enable you to maintain that aura of mystique that surrounds all programmers.

PROGRAM ORGANIZATION

The entire program for playing *Xchange* is illustrated by the template in Fig. X-1. Each of the nine blocks in the illustration has a logical (as opposed to physical) connotation.

Task Orientation

The task orientation of each of the nine blocks is as follows:

- **Initialize.** Program housekeeping logic, including an optional call to print the game's description.
- **Instructions.** This is a linear series of print statements constructed with quoted alpha strings. Each statement generates one line of output on the primary display device.
- **# Players?** The dialog for selecting either the single or dual mode is contained in the # Players? sequence. Response qualification and a keyboard error message is included here also.
- **Scrambler.** It shuffles into random order a pair of strings (A through H and an asterisk). The result is stored in two tables, one for each player. The print grids routine is called from here once to display the scrambler's handiwork.

- **Player X.** This is the solitaire player's input routine. It is also, in the dual-player mode, the input logic for the player whose grid is on the left (of the two grids). This routine is self-sufficient. It prompts the player, accepts and fully qualifies the response, and does the exchange business for this player's grid.
- **Player Y.** Logically this routine is an exact duplicate of the other player's. It does address its own string table, and all message positioning is tabbed so as to appear to the right of the other's grid. The TAB values correspond to the base-print positions for this player's own grid.
- **Print Grids.** This module of coding outputs the contents of the *X* and *Y* tables. It includes a conditional test to suppress the printing of the second grid if the program is running in the solitaire mode.
- **Win Check.** The scrambler module contains a DATA series of A, B, C, D, E, F, G, H, *, which is supposed to be the winning sequence. This routine reads that string and compares it to the player's tables. A looping process is used and the first instance of an out-of-order condition terminates the test (per player). If the loop succeeds in running nine times a winner is declared.
- **Mainline.** This is a task dispatcher. It calls each of the supporting subroutines in a reiterative fashion. There are conditionals included to avoid callups of routines that are not needed when only one player is involved. This routine also contains the logic that permits an early termination by testing for an X keyboard character. At the bottom of this module is the code-sequence that handles the end-of-game and the replay option.

Salient Points

The recipe for *Xchange* is shown in the program's listing. All of the preceding descriptions are assumed to be comprehensive enough, so let us bump down through the listing, pointing to the more interesting details only.

In the housekeeping area, at lines 40, 50, and 60, some alpha messages are initialized into variables denoted with an *M*. There are several places in the program where these messages are apt to be needed. Canning them in an alpha variable permits their printing by symbolic reference rather than having to duplicate quoted print expressions. Notice also an asterisk character is prestored in a similar manner by statement 70.

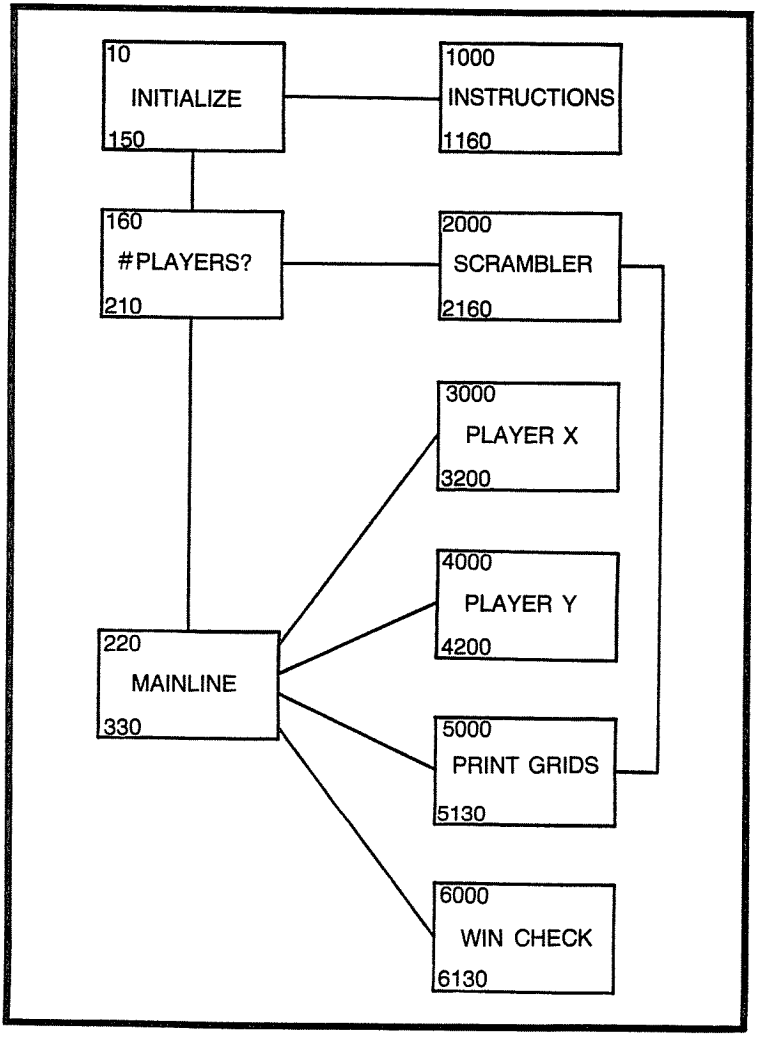


Fig. X-1. Program template for Xchange.

Statement 80 has the dimensioning expressions for the *X* and *Y* tables, and is followed by the conditional dialog for printing the game's instructions at the player's option. Whether the instruction module is called or not, serial flow resumes at line 160 to accept a code of one or two to indicate the number of players.

Notice how emphatic this sequence is. Statements 180 and 190 explicitly test for an acceptable code. Anything other than a one or a two is rejected by the branch in line 2000 and a retry is enforced. This is an example of programming convenience. Later, when it is

desired to condition the program's logical flow based on the player number, there are only two possibilities in *P*, absolutely.

Task dispatching is accessed next—this is the mainline—and it extends from line 210 to 270. Within this area, in line 225, is the "easy out" if an X is typed at the player X entry point. The following statement (line 230) is one of the several that controls whether one or two playing grids are provided for. Otherwise, the mainline consists of GOSUB statements for ordering the jumps to the supporting routines.

Every lap through the mainline area does include the jump to line 6000, where the test is made to see whether either string has been sorted out. Upon returning from the win check subroutine, the *I* variable, which was used by the testing loop, may or may not have a residue of eight. If it does, the loop did run until done, meaning one or the other of the player's strings are properly ordered so the mainline's looping should come to an end. (See the series from line 270 through 330 for the end-of-game options.)

The call to line 2000 from statement 210 is only accessed the first time the mainline is executed. That call to the scrambler sets up the strings initially; so it is apt to glance there next to see how it works.

The scrambler uses three FOR-NEXT loops. The first is a READ task which loads the DATA string into *table X*. Notice the restore instruction in line 2040; this will reset the READ pointer so that this module will work correctly if another game is called for. The win check loop uses the DATA from here as well. The prevailing design convention is: whoever last did a READ is responsible for doing a restore as well.

The second phase of the scrambling process uses an RND function to shuffle the contents of *table X*. Statements 2050 through 2110 comprise the whole of this phase. This loop runs nine times too. Each of the elements is picked up serially and exchanged with one of the others (including itself, possibly) based on the RND subscript. When the loop bottoms out, phase 3 of the scrambler begins.

With *table X* completely shuffled all that is needed to support the dual-player mode is to generate *table Y*. The loop from line 2120 to line 2140 does this by merely copying the contents of *X* into *Y* a line at a time. This replication is relatively fast; so it is always done even though the game may be running in the solitary mode. It seems doubtful that any lone player would be fully appreciative of the milliseconds that could be saved if this loop were conditionally bypassed.

Anyway, the scrambler finishes with a jump out of line 2150 to print either one or two grids, depending on the mode, and a relink to the mainline will get the game under way. Notice that there is a nested pair of subroutines here. The return at the conclusion of the print grids will come back to line 2160, and that return will send the program all the way back to line 220.

The GOSUB 3000 is to the player X routine. The single word MOVE is output by statement 3000 to prompt the player to type something. At this point, if an X is entered into *Q$*, the conditional expression in line 3015 will bypass the balance of this module by going directly to the RETURN in line 3200. Any value other than an X will be regarded as normal and an attempt will be made to accomplish the requested exchange.

The convention for expressing a move choice is for the player to type the letter that he or she wishes to be exchanged with the asterisk. In the event the player is confused or careless, and instead types an asterisk itself, the logic of statement 3020 will default to print an "ILLEGAL MOVE" note. From there the branch in line 3040 will execute to permit another entry. This test is made as an independent step so that a loop may be used to scan-compare the value of *Q$* with the values in the string table.

The loop that begins in line 3050 and runs until line 3070 is looking to find a match, meaning that whatever was typed is at least a valid letter value. There is a usefulness to the position counter at which a match is found also, as will be shown momentarily.

Meanwhile, if the loop searches the whole table without finding any match (the asterisk in the table is skipped over), whatever was typed is unintelligible; so an "INVALID CHARACTER" message is printed. Although there may be some semblance of validity to the character if a match is found, another rule must be qualified. Is the letter adjacent to the asterisk? The logic from line 3130 to line 3160 checks this, again by presuming a good code exists. If these tests do fail the default route includes an error note and a branch back for another entry. This test is critical and here's why.

The brief loop that is stated in lines 3100 through 3120 will generate into the *J* variable the current table location of the asterisk. Because *I* has not been tampered with the location of the player's choice for an exchange is also known. A simple arithmetic comparison of *I* and *J* will determine whether the asterisk is adjacent to the letter indicated for an exchange. If *I* and *J* differ by only one they are adjacent horizontally. If the difference is exactly three they appear to be adjacent vertically. If any of the explicit tests are satisfactory it is safe to do the exchange.

Only the two statements in lines 4180 and 4190 are needed to swap the move choice with the asterisk. It doesn't matter at all which of the four THEN functions got us here. The program can't get this far unless the player's move is programmatically acceptable. After the move is made, exit player X.

Returning to the mainline briefly: is player Y needed? If the conditional in line 230 is defaulted the GOSUB 4000 will accommodate the second player. Talk about redundancy. Nearly all of the lines from 4000 to 4200 are duplicates of lines 3000 to 3200, line for line. *Table Y* is worked on by this routine instead of the first player's string, and all of the line references are in the 4000 series of course. Another minor modification is contained in lines 4000 and 4090 as well. The TAB(16) keeps the printed output aligned to the right of the first player's output area.

Perhaps a philosophical excuse is in order here for simply copying twenty-one lines of the program internally. A single routine with dual-conditioning could have been designed to be shared by both players. Agreed. But would the savings in lines of coding be worth it? What is the likelihood that the increased complexity would hamper debugging? Anyway, to me, KISS seemed to be a good idea. Although neither you nor I are particularly stupid, remember: "Keep it simple . . . . " End of argument.

The next jump out of the mainline is to line 5000 to print the grids. Nested FOR-NEXT loops can at times be a wee bit complex, but this case isn't especially so with a little study. The outside loop (*I*) is set up in line 5010. Incrementing this variable by three will cause a FROM series of 1, 3, 6, and so on. The TO limit of nine will allow the first loop to run only three times. Three print lines are needed.

Three characters are needed for each line, also. The inside loop is conditioned from the current value in *I*, and it uses *J* as its own counter. The conditional in line 5050 checks whether two playing grids are supposed to be printed. If so, as each line of three characters on the left is completed—meaning one completed revolution of the *J* loop for three times—before the next left-hand print is done another *J* loop (lines 5100 through 5130) is used. This minor redundancy is excused by the need to reference the other table (*Y*) and by the need for a print statement with a TAB.

PRINT GRIDS

Actually, the only thing tricky about the print grids routine is controlling the line spacing. Particular care must be taken in the use of the trailing semicolons in the print expressions and in the use of

the seemingly cosmetic print functions. When PRINT stands alone here it has logical significance: a line is completed and it's time to space up one line in the printed output. Once the looping and the printing are done it is time to get back to the mainline. Reenter line 260.

This last jump— a GOSUB 6000—is the last for you, me, and the mainline. So simple it is, too. Both tables are checked by independent FOR-NEXT loops that include a READ to compare the DATA from the scrambler module (see line 2000) with each element in the tables. If either loop can get as far as finding out that the bottom spot must have the asterisk, and that at the count of eight everything above is in order, a terse note is output declaring the winner. Notice that *table Y* is tested conditionally, based on the results of statement 6070. In most microcomputer implementations of BASIC a reiterative process can be discernibly slower than for some other techniques, albeit only marginally so sometimes.

THE END

One way or another this routine will ultimately finish via the return in line 6060. Back in the mainline any residue of eight in the *I* counter is sufficient notice to the program that someone won. (Who it was has already been noted in the print area.) If so the game is over. If not the junk value in *I* will cause the mainline loop to go again.

And you can believe, for some players anyway, it seems they will go on forever. That is why the game of *Xchange* can be fun. Try it yourself and you may find this recipe affords a dish that can cause mental indigestion. Remember too, like an antacid, the X cure is only for temporary relief.

THE PROGRAM

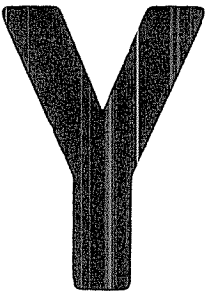
```
 10 REM "XCHANGE"
 20 REM
 30 GOSUB 9000
 40 LET M1$ = "INVALID CHARACTER"
 50 LET M2$ = "ILLEGAL MOVE"
 60 LET M3$ = "YOU'RE THE WINNER"
 70 LET A$ = "*"
 80 DIM X$(9), Y$(9)
 90 PRINT "WANT INSTRUCTIONS (Y OR N)";
100 INPUT Q$
110 IF Q$ = "N" THEN 160
120 IF Q$ = "Y" THEN 150
```

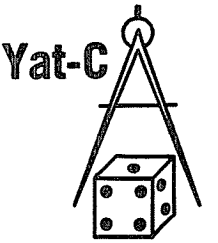
```
130 PRINT "HUH?"
140 GOTO 90
150 GOSUB 1000
160 PRINT "NUMBER OF PLAYERS (1 OR 2)";
170 INPUT P
180 IF P = 1 THEN 210
190 IF P = 2 THEN 210
200 GOTO 160
210 GOSUB 2000
220 GOSUB 3000
225 IF Q$ = "X" THEN 280
230 IF P <> 2 THEN 250
240 GOSUB 4000
250 GOSUB 5000
260 GOSUB 6000
270 IF I <> 8 THEN 220
280 PRINT
290 PRINT "ANOTHER GAME (Y OR N)";
300 INPUT Q$
310 IF Q$ = "Y" THEN 160
320 PRINT "SO LONG ..."
330 END
1000 PRINT "1 OR 2 MAY PLAY."
1010 PRINT "IF 2, YOU TAKE TURNS."
1020 PRINT "A GRID LOOKS LIKE THIS:"
1030 PRINT " F G D"
1040 PRINT " A H *"
1045 PRINT " E B C"
1050 PRINT "BUT IT SHOULD LOOK LIKE THIS:"
1060 PRINT " A B C"
1070 PRINT " D E F"
1080 PRINT " G H *"
1090 PRINT "YOU MAY EXCHANGE ANY 1 LETTER"
1100 PRINT "WITH THE * - BUT ONLY 1 THAT'S"
1110 PRINT "ADJACENT: ABOVE, BELOW, LEFT,"
1120 PRINT "OR RIGHT."
1130 PRINT "READY";
1140 INPUT Q$
1150 PRINT "HERE WE GO ..."
1160 RETURN
1999 REM  "SCRAMBLER"
2000 DATA A, B, C, D, E, F, G, H, *
2010 FOR I = 1 TO 9
```

```
2020 READ X$(I)
2030 NEXT I
2040 RESTORE
2050 FOR I = 1 TO 9
2060 LET R = INT(10*RND(1))
2070 IF R < 1 THEN 2060
2080 LET X$ = X$(I)
2090 LET X$(I) = X$(R)
2100 LET X$(R) = X$
2110 NEXT I
2120 FOR I = 1 TO 9
2130 LET Y$(I) = X$(I)
2140 NEXT I
2150 GOSUB 5000
2160 RETURN
2999 REM "PLAYER-X MOVES"
3000 PRINT "MOVE";
3010 INPUT Q$
3015 IF Q$ = "X" THEN 3200
3020 IF Q$ <> A$ THEN 3050
3030 PRINT M2$
3040 GOTO 3000
3050 FOR I = 1 TO 9
3060 IF Q$ = X$(I) THEN 3100
3070 NEXT I
3080 PRINT M1$
3090 GOTO 3000
3100 FOR J = 1 TO 9
3110 IF X$(J) = A$ THEN 3130
3120 NEXT J
3130 IF I+1 = J THEN 3180
3140 IF I+3 = J THEN 3180
3150 IF I-1 = J THEN 3180
3160 IF I-3 = J THEN 3180
3170 GOTO 3030
3180 LET X$(J) = X$(I)
3190 LET X$(I) = A$
3200 RETURN
3999 REM "PLAYER-Y MOVES"
4000 PRINT TAB(16) "MOVE";
4010 INPUT Q$
4020 IF Q$ <> A$ THEN 4050
4030 PRINT M2$
```

```
4040 GOTO 4000
4050 FOR I = 1 TO 9
4060 IF Q$ = Y$(I) THEN 4100
4070 NEXT I
4080 PRINT TAB(16) M1$
4090 GOTO 4000
4100 FOR J = 1 TO 9
4110 IF Y$(J) = A$ THEN 4130
4120 NEXT J
4130 IF I+1 = J THEN 4180
4140 IF I+3 = J THEN 4180
4150 IF I-1 = J THEN 4180
4160 IF I-3 = J THEN 4180
4170 GOTO 4030
4180 LET Y$(J) = Y$(I)
4190 LET Y$(I) = A$
4200 RETURN
4999 REM  "PRINT GRIDS"
5000 PRINT
5010 FOR I = 1 TO 9 STEP 3
5020 FOR J = I TO I+2
5030 PRINT " "X$(J);
5040 NEXT J
5050 IF P = 2 THEN 5100
5060 PRINT
5070 NEXT I
5080 PRINT
5090 RETURN
5100 FOR J = I TO I+2
5110 PRINT TAB(16) " "Y$(J);
5120 NEXT J
5130 GOTO 5060
5999 REM  "WINNER CHECK"
6000 FOR I = 1 TO 8
6010 READ X$
6020 IF X$(I) <> X$ THEN 6070
6030 NEXT I
6040 PRINT M3$
6050 RESTORE
6060 RETURN
6070 IF P <> 2 THEN 6050
6080 FOR I = 1 TO 8
6090 READ Y$
```

```
6100 IF Y$(I) <> Y$ THEN 6050
6110 NEXT I
6120 PRINT TAB(16) M3$
6130 GOTO 6050
9000 REM  "RANDOM NUMBER ROUTINE"
9010 LET Z = RND(1)
9020 RETURN
```





For those in the know here is another obvious takeoff of a commercial board game. Whether you recognize the traits and the departures from Yahtzee or not, it is the program that drives this game that is worth studying.

The game is played by two persons taking turns, rolling five dice. A game is over when either passes 500 points; high score wins. In turn each player is given three chances to better his or her hand before scoring the round. The options are to pass, hold on to any one number and roll the other four, or to roll a particular number, leaving four as they lie. The choice in each case depends on a knowledge of how score is derived and an intuitive feel for the odds of what is apt to happen in another roll.

Straights are hard to fill, as any poker player will tell you. In this game a one through five series or two through six is worth 200 points. Matches are scored by multiplying the number in a set by the total of the five dice. For example, assume the series 4-1-5-4-4 remains after a player's third roll. The score for this turn is fifty-four. (There are three fours; the sum of the five numbers is eighteen; three times eighteen is fifty-four.) There is one other possibility: no straight and no matches. The program calls this hand a natural. The score for a natural is simply the sum of the five dice.

A full house exits when two dice have one value and the remaining three have another single value. In this case the score is twice the sum of the pair and thrice the sum of the triplets.

All of these mechanics may be easily changed. This program is fun to have in your library as-is, and it provides opportunities for

experimenting with end-of-game limits and scoring points permutations. As implied earlier these are not the only reasons for including this design in this book.

The regular position of *Y* as next to the last affords an opportunity to contrast a complex design with the dramatic simplicity that will be found in Chapter *Z*. The complexity of *Yat-C* is not in its procedural logic; rather, it is in its structural organization. In some schools this design would come close to being described as over engineered.

That is why there is opportunity here to extol, a last time, my propensity for the template method of documenting BASIC programs. At a glance the picture in Fig. Y-1 may in fact look like a

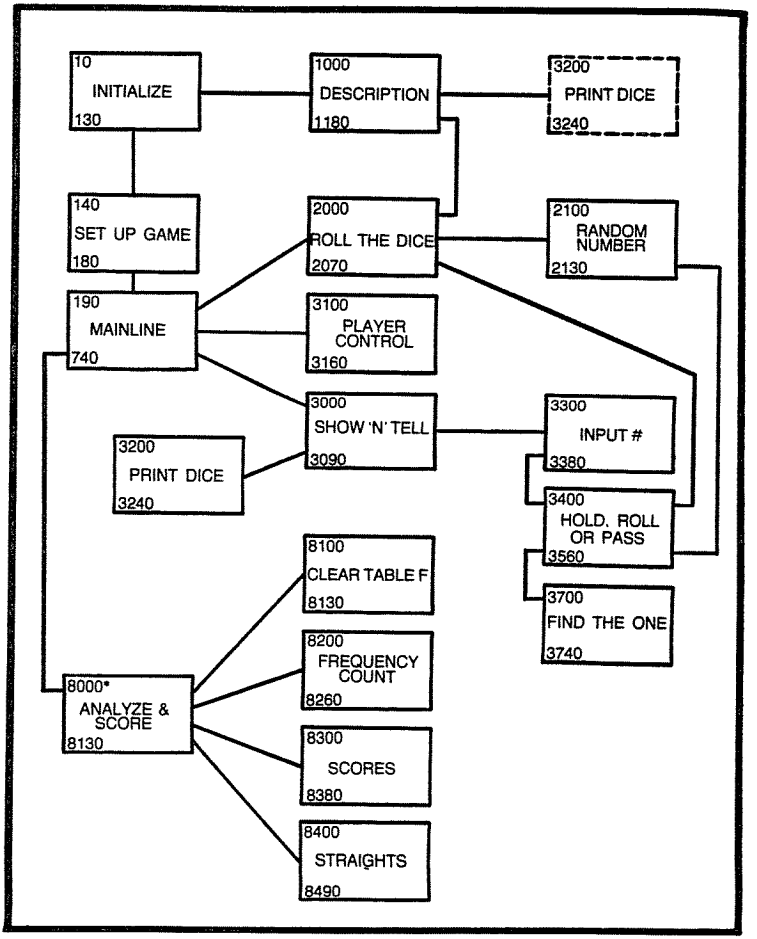


Fig. Y-1. Program template for Yat-C.

plumber's nightmare. Yet, with the briefest study a sense of reason can be gleaned from it.

Not much time has been spent elsewhere on how templates are formed, so it is here that this deficiency can be corrected. It is hoped that you will find benefit in this form and the ease with which complex programs can be documented. In any event a few minutes perusal should prove worthwhile for novitiate programmers. Even seasoned coders are apt to appreciate an old trick included, using subroutines that call themselves.

THE PROGRAM TEMPLATE

Most of the fun in programming has to do with solving a certain problem. Nearly all schools preach a discipline that insists upon some form of structuring of the whole job from the outset, including all of those rote tasks that are in large part pure repetition. On big advantage of the template scheme is that it will tolerate our beginning the job in the area where it is the most fun: yet the finished program will appear to have been orderly conceived and implemented.

The template idea, as used throughout this book, is not suggested as a replacement for traditional methods of programming and documentation. I do suggest, however, that the template can serve two roles: one in front of, and the other after the fact. What follows is a succinct narrative of how we build the template and the program, using the first to aid the second. When finished the picture of the program is what is left over. Here is how.

The buildup of *Yat-C* supposed several programming tasks were needed from the outset. A first cut of the design was sketched as shown in Fig. Y-2. In this pass my thinking was along these lines:

- All programs need initialization to at least some extent
- Repeated play should be an option: a set up game sequence will be necessary
- A mainline area can be used to hook together whatever subroutines are finally built
- Any instructions that are needed would be assigned to block 1000
- The dice-rolling sequence would be assigned to block 2000
- The needed player input would be assigned to block 3000
- In coding the tricky part is to let the player hold, roll, or pass. By coding this as a subroutine, starting at say 3400, other tasks associated with player input can be done in the 3000 series also

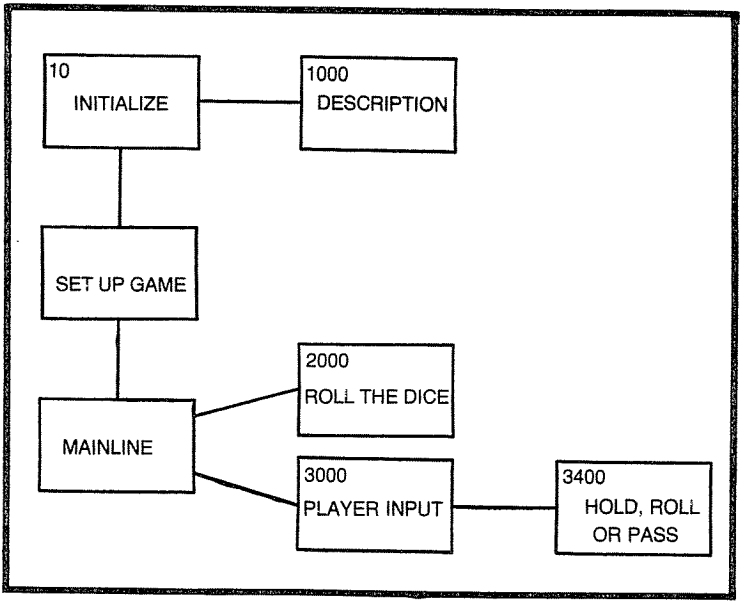


Fig. Y-2. The first-cut template for Yat-C.

Almost as soon as my pencil touches the paper it is realized that input will be needed, but there is no real fun in doing all of that dialog business and keyboard error checking. That can wait. At this point just code in a GOSUB. Because the keyboard entries will have to occur before this routine (line 3400) can be used, line 3300 is arbitrarily used as the eventual address for that module. Carry on.

And so the programming continues. Each time impatience causes me to want to wait until later to do the detail coding for a specific task, I simply pick a block number, sketch the block on the drawing, and continue with the gritty part at hand.

When finished, backing up all the while, finally doing the lesser tasks, the drawing will show it all, even if somewhat helter-skelter. But it is all there, and it only takes a few minutes to copy it, repositioning the blocks so that lines can be drawn connecting them without ever having to cross over each other.

The general philosophy regarding the connecting lines is that lines connected to the top or bottom of a block represent straightthrough flow. Subroutine calls and return paths are depicted by connecting the blocks with lines into or out of the sides of the blocks.

Two problems can occur while drawing the template for a very complex program (or for a very helter-skelter program, as the case

may be). In Fig. Y-1 the block drawn with broken lines (the coding series spanning 3200 through 3400) is duplicate of another block. The subroutine to print dice exists in the program only once. The problem, and the reason for showing the block twice, is that to connect the description block to the solid-line block would require crossing over other connecting lines. I would really rather not have to.

The other type of problem is that one associated with block 8000 in Fig. Y-1. The solution in this case is to append a footnote flagged with an asterisk. The nature of the problem here is that which was mentioned earlier: while within a jump to line 8000 another jump is executed, jumping a few lines later to an area that will be executed again (but in a straight-line manner the second time). In effect this is a subroutine that calls the tail end of itself, then bottoms out through the same (final) return.

Naturally, the blocks and the lines by themselves do not convey the complete picture. A sense of sequence depends on a couple of other template conventions. Whenever possible the blocks should be positioned in a top-to-bottom order according to their calling sequence. That is why, in Fig. Y-1, line 3100 is positioned above line 3000.

The names shown inside the blocks attempt to convey order-of-use also. These same names are used in REM statements in the program as well. This and the use of last-line numbers in the blocks tends to tie the template and the program together. The combination works to give adequate documentation, but with little real effort and without detracting from the fun of programming.

THE CODING OF YAT-C

A few moments thought about what it is this program has to do will predict that not a lot of storage is required. That is the case. Two small tables are set aside in line 40. *Table D*, with five elements, is for the five dice. That six-element table called *F* is for frequency counting. (There are six numbers possible on any one die. A pair of sixes would cause a two in the sixth position of *table F*, for example.)

All other storage is in simple variables, and even they are few in number. Four of these are initialized in line 150 through 180. Symbol *S* is for score; this is a temporary worker, used for either player during his or her turn. Results accumulated in *S* are added to *S1* or *S2*, the total-score workers (one for each player). Players are also known by a number, either *1* or *2*, and the variable *P* shows which player is up.

Recurring rounds of play are driven by the sequence in the mainline that extends from line 190 through 600. The brief FOR-NEXT loop in lines 190 through 210 is for clearing of the five dice workers at the start of each round. This is done because the jump to line 2000 is for rolling five (new) numbers into the *table D* fields.

The logic of the dice roller (lines 2000 through 2070) is such that it examines each spot in *table D*, inserting a newly generated random number in any location that has a zero in it (but leaving as-is any spot already containing a number). From this it is easy to guess how the "hold or roll" option works—to fetch any new number, just zero the spot and GOSUB 2000.

This is an apt place to notice that the random-die function is set aside as a separate subroutine, callable from a variety of locations. It is in lines 2100 through 2130, and simply returns in *R*, a number from one to six.

A key jump out of the mainline is to line 3100. It is there (lines 3100 through 3160) that the player number (*P*) is alternated from *1* to *2*, and the players' scores are updated by whatever residue is in *S* following each completed round of play (after all three of a player's rolls). It is because this jump occurs first, before any player input at all (at the beginning of a game) that *S* must be preconditioned to zero, and *P* is started up with a two in it.

The next jump out of the mainline is to line 3000. That subroutine is in concept a second mainline. The list of alternating GOSUB statements from line 3000 to line 3090 causes a continuous print-enter-print type of sequence. The reason for isolating the print dice task is that a round for a player begins and ends with printed output. Printing is done four times; input is done only three.

The dice printer is pure simplicity: FOR-NEXT loop is used, outputting the five numbers stored in *D(1)* through *D(5)*. The other jump, to line 3300, is for player input. The total sequence from line 3300 to line 3380 is not long either, but it contains other jumps, triggering a long line of nested accesses.

All that is asked for in lines 3300 through 3340 is a number ranging from zero to six. Whatever is typed is accepted into *Q*, and if valid it is carried along by the jump to line 3400, which is where I started coding this program.

As seen by the player, to this point he or she has been shown five dice, asked for the number on one of them, and is now prompted with "(H, R, OR P)", followed by a question mark. From line 3420 to line 3440 the program looks at what was typed into *Q$*, and acts accordingly.

If the player likes what is seen, and figures to roll anything is more apt to cost than help, a P (for pass) response will end the turn.

The exit logic goes to line 3525 for the return back to the minimainline.

If the player's option is to hold the sequence from line 3470 to line 3525 comes into play. This sequence interprets the hold option to mean that the number being held in *Q* (supposedly matching one of the dice showing) is to be kept, and the other four numbers are to be newly generated. The jump out of line 3470 to line 3700 is for two purposes: to determine the number in *Q* is valid, and if so, to ascertain its *table D* location. (In cases where the number occurs more than once the first will suffice; either way *Q1* will tell.)

That test for zero in *Q* in line 3475 was patched in to exit from here back to the number-entry area to force another try (*Q* wasn't found in *table D*). If the number in *Q* is good, the FOR-NEXT loop from line 3480 to line 3500 clears the whole dice table. Then line 3510 reloads the *Q1* location with *Q* in effect at whatever position it was found. The jump from line 3520 to line 2000 will refill the four zeroed spots with new numbers, and the round goes on.

The remaining option is *R* (for roll). Starting at line 3530 the program's logic presupposes that the player is happy with four of the five dice and would like to selectively roll one by itself. Again, we jump to line 3700 to find out which number matches *Q* (or whether a keyboard mistake was made). This one is easy. Line 3540 jumps to line 2100 to get a number, and it is then superimposed over whatever was in the table where *Q* was found. And the round goes on.

When the minimainline is exhausted the program returns to the mainline at line 500. This last jump from the continuous-play loop is to line 8000, where we have another minimainline, this one for score calculations.

Calculation of a player's score begins with a jump to line 8100. The short FOR-NEXT loop from line 8100 to line 8130 clears the frequency count table. The return in line 8130 goes back to line 8020 to pick up the jump to line 8200. This second task, from line 8200 to line 8260, loops and counts the pairs and so on that are in the five dice workers, loading the results into *table F*.

Now, the heavy: GOSUB 8300. A six-times-always loop from line 8302 to line 8370 is used to test each position in *table F*. Each of the possible numbers (two through five) are checked for. The *S$* variable is loaded with an appropriate message as this loop goes on. An exception condition is managed by use of an *F* counter within the loop.

If a two is found *F* is incremented to one. As the loop continues, if a three is also found, *F* is again incremented. (Notice the test for another "YAT" in line 8305; no extra score is given for a second

pair.) When the loop finally completes, if *F* is equal to two, *S$* is overwritten with the *full house* constant. For arithmetic reasons, it doesn't matter which comes first, a pair or triplets—the compounded score will be the same.

By the way, if no counter has anything greater than a one in it, the natural in *S$* will remain unchanged and *S* is not affected. A straight might be in *table D*, though, and that is what is checked next.

In line 8400 if the first and the last positions in the frequency table are alike get out quick. With only five dice, a straight cannot embrace one *through* six. From line 8420 to line 8450 the loop looks for zeros. The logic of this routine depends on the fact that only one zero can be there.

The fallthrough flow back down through lines 8100 to 8130 reclears *table F* as a last act in calculating scores. A return could be inserted in line 8055, but had this been done I would have missed the chance to demonstrate the *subroutine that calls part of itself*. And that would have removed some of the fun.

THE PROGRAM

```
 10 REM  "YAT-C"
 20 REM
 30 GOSUB 9000
 40 DIM D(5), F(6)
 50 PRINT "WANT THE DESCRIPTION (Y OR N)";
 60 INPUT Q$
 70 IF Q$ = "N" THEN 140
 80 IF Q$ = "Y" THEN 130
 90 PRINT "ANSWER WITH Y (FOR YES), OR"
100 PRINT "ANSWER WITH N (FOR NO)."
110 PRINT
120 GOTO 50
130 GOSUB 1000
140 PRINT
150 LET S = 0
160 LET P = 2
170 LET S1 = 0
180 LET S2 = 0
190 FOR I = 1 TO 5
200 LET D(I) = 0
210 NEXT I
300 GOSUB 2000
310 IF S1 > 500 THEN 700
```

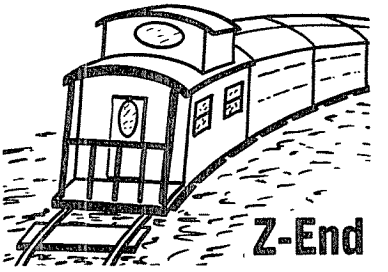
```
320 IF S2 > 500 THEN 700
330 GOSUB 3100
400 GOSUB 3000
500 GOSUB 8000
600 GOTO 190
700 GOSUB 3140
705 PRINT "WANNA PLAY AGAIN (Y OR N)";
710 INPUT Q$
720 IF Q$ = "Y" THEN 140
730 PRINT "GOOD-BYE, THEN..."
740 END
999 REM  "DESCRIPTION"
1000 PRINT "THIS IS A COOL 2-PLAYER GAME."
1010 PRINT "EACH TURN IS 3 ROLLS OF 5 DICE"
1020 PRINT "LIKE THIS:"
1030 GOSUB 2000
1040 GOSUB 3200
1050 PRINT
1060 PRINT "AFTER THE 1ST ROLL, YOU MAY:"
1070 PRINT "'H'  HOLD 1 NUMBER AND ROLL THE"
1080 PRINT "     OTHER 4, OR YOU MAY"
1090 PRINT "'R'  ROLL ANY 1 NUMBER AND"
1100 PRINT "     HOLD THE OTHER 4 OR YOU MAY"
1110 PRINT "'P'  KEEP WHAT YOU HAVE (PASS)."
1112 PRINT "OK";
1114 INPUT Q$
1120 PRINT "SCORING: STRAIGHT = 200 POINTS"
1130 PRINT "YAT-C (5/KIND) = 5 X DICE VALUE"
1140 PRINT "YAT-B (4/KIND) = 4 X DICE VALUE"
1150 PRINT "YAT-A (3/KIND) = 3 X DICE VALUE"
1152 PRINT "YAT   (2/KIND) = 2 X DICE VALUE"
1154 PRINT "FULL HOUSE = 2X + 3 X DICE VALUE"
1160 PRINT "NATURAL (MIXED BAG) = VALUE OF"
1170 PRINT "THE DICE ONLY."
1180 RETURN
2000 REM  "ROLL THE DICE"
2010 FOR I = 1 TO 5
2020 LET R = D(I)
2030 IF D(I) <> 0 THEN 2050
2040 GOSUB 2100
2050 LET D(I) = R
2060 NEXT I
2070 RETURN
```

```
2099 REM  "A RANDOM NUMBER"
2100 LET R = INT(10*RND(1))
2110 IF R < 1 THEN 2100
2120 IF R > 6 THEN 2100
2130 RETURN
3000 REM  "SHOW 'N' TELL"
3020 GOSUB 3200
3030 GOSUB 3300
3035 IF Q$ = "P" THEN 3090
3040 GOSUB 3200
3050 GOSUB 3300
3055 IF Q$ = "P" THEN 3090
3060 GOSUB 3200
3070 GOSUB 3300
3075 IF Q$ = "P" THEN 3090
3080 GOSUB 3200
3090 RETURN
3099 REM  "PLAYER CONTROL"
3100 IF P = 1 THEN 3125
3105 LET P = 1
3110 LET S2 = S2+S
3115 IF S = 0 THEN 3150
3120 GOTO 3140
3125 LET P = 2
3130 LET S1 = S1+S
3140 PRINT
3145 PRINT "SCORES: #1" S1 " #2" S2
3150 PRINT
3155 PRINT "PLAYER #" P
3160 RETURN
3199 REM  "PRINT DICE"
3200 PRINT "  ";
3210 FOR I = 1 TO 5
3220 PRINT D(I);
3230 NEXT I
3240 RETURN
3299 REM  "INPUT #"
3300 PRINT "NUMBER";
3310 LET Q = 0
3320 INPUT Q
3330 IF Q < 0 THEN 3370
3340 IF Q > 6 THEN 3370
3350 GOSUB 3400
```

```
3355 IF Q = 0 THEN 3370
3360 RETURN
3370 PRINT "HUH?"
3380 GOTO 3310
3399 REM  "HOLD, ROLL, OR PASS"
3400 PRINT "(H, R, OR P)";
3405 LET Q$ = " "
3410 INPUT Q$
3420 IF Q$ = "H" THEN 3470
3430 IF Q$ = "R" THEN 3530
3440 IF Q$ = "P" THEN 3525
3450 PRINT "OOPS - H, R, OR P"
3460 GOTO 3405
3470 GOSUB 3700
3475 IF Q = 0 THEN 3520
3480 FOR I = 1 TO 5
3490 LET D(I) = 0
3500 NEXT I
3510 LET D(Q1) = Q
3520 GOSUB 2000
3525 RETURN
3530 GOSUB 3700
3535 IF Q = 0 THEN 3560
3540 GOSUB 2100
3550 LET D(Q1) = R
3560 RETURN
3699 REM  "FIND THE ONE"
3700 FOR Q1 = 1 TO 5
3710 IF D(Q1) = Q THEN 3740
3720 NEXT Q1
3730 LET Q = 0
3740 RETURN
7999 REM  "ANALYZE & SCORE"
8000 PRINT "      ";
8010 GOSUB 8100
8020 GOSUB 8200
8030 GOSUB 8300
8040 GOSUB 8400
8050 PRINT S$
8099 REM  "CLEAR TABLE-F"
8100 FOR I = 1 TO 6
8110 LET F(I) = 0
8120 NEXT I
```

```
8130 RETURN
8199 REM  "FREQUENCY COUNTS"
8200 LET S = 0
8210 FOR I = 1 TO 5
8220 LET J = D(I)
8230 LET F(J) = F(J)+1
8240 LET S = S+J
8250 NEXT I
8260 RETURN
8299 REM  "SCORE"
8300 LET S$ = "NATURAL"
8301 LET F = 0
8302 FOR I = 1 TO 6
8304 IF F(I) <> 2 THEN 8310
8305 IF S$ = "YAT" THEN 8310
8306 LET S$ = "YAT"
8308 LET S = S+2*S
8309 LET F = F+1
8310 IF F(I) <> 3 THEN 8330
8315 LET S$ = "YAT-A"
8316 LET F = F+1
8320 LET S = S+3*S
8330 IF F(I) <> 4 THEN 8350
8335 LET S$ = "YAT-B"
8340 LET S = S+4*S
8350 IF F(I) <> 5 THEN 8370
8355 LET S$ = "YAT-C"
8360 LET S = S+5*S
8370 NEXT I
8372 IF F < 2 THEN 8376
8374 LET S$ = "FULL HOUSE"
8376 LET F = 0
8380 RETURN
8399 REM  "STRAIGHTS"
8400 IF F(1) = F(6) THEN 8490
8410 LET F = 0
8420 FOR I = 1 TO 6
8430 IF F(I) <> 0 THEN 8450
8440 LET F = F+1
8450 NEXT I
8460 IF F <> 1 THEN 8490
8470 LET S$ = "STRAIGHT"
8480 LET S = 200
8490 RETURN
9000 REM  "RANDOM NUMBER ROUTINE"
9010 LET Z = RND(1)
9020 RETURN
```

Z



This is *the* end. If you lisp a little it might even sound like Z-end. A corny name, perhaps, but chosen because of a propensity for rounding out our library with a simple classic. The game is based on the ancient game of nim.

The word *nim* is itself archaic, and means roughly to steal, filch, or take away. That is how this game called *Z-End* is played. Players take turns taking away letters of the alphabet. The one that gets the *Z* ends the game and is the loser.

Simple game. Simple program. Included because it is a classic, but more importantly because it has a definite usefulness.

Need to demonstrate to a friend how easy it is to program? This is a good model. It includes alphanumeric variables, conditional logic, elementary arithmetic, DATA structures, and even FOR-NEXT loops. Its design is basically *straight-line* and therefore simple. And it shows in a rudimentary way how the computer is able to play a game.

This is also a good game for introducing new players to your computer. Young and old alike will enjoy the ease with which they can learn to play with your machine. After a few rounds, and they move on to other games, they will surely remember the fun they had in *Z-End*.

FROM THE TOP

There is an option provided by lines 40, 50, and 60 to skip the rules. Notice that, unless explicitly skipped with a *Y*, the rules will

be printed. In many programs both *Y* and *N* are emphatically checked, overcoming any keyboard accidents by allowing another try. The choice as to when to use this easier programming method is not capricious.

Rules

Since the rules option is permitted but once, right after the program is loaded, an operator's mistake at this point may let him or her enter into a complex game without any knowledge as to how it is played. *Z-End* is not risky. It is a simple game, and only a few rounds are needed to conclude a trial round as the learning process. Thus, there is not any real harm done if the rules question is answered too hastily. Anyway, *Y* is all that is checked for in this model.

Housekeeping

A game's housekeeping is done from line 160 through 185. The DATA string containing the alphabet is declared here, to be used later. The symbol *Z* is set to one. Later, each draw is added to *Z*, and when it passes twenty-five the game is over. Dual-purpose worker *Q* is zeroed at this point because the GOSUB 500 uses it in an arithmetic expression. When the game first starts no residue should be left in *Q* from a previous game.

Printout

The subroutine that extends from line 500 down to line 660 is the section of coding that prints the alphabet. From line 550 to 580 is a brief FOR-NEXT loop that reads the DATA string, stopping with whatever letter should be the first in the forthcoming round of play. In the beginning, of course, the loop will break immediately because the conditional in line 570 will find that *Z* does contain a one.

Rounds of play

Another FOR-NEXT loop is entered into from the first one (lines 600 through 620). This one begins with whatever has been read into *Z$*, prints it, and continues to read and print until it reaches the end of the alphabet. When the *restore* is finally reached in line 630 the DATA pointer is reset so that succeeding rounds will begin again with *A* in the alphabet. The null print in line 640 advances the printed output, and the conditional in line 650 will let the return work if there is more than one letter left in the string.

"YOUR TURN" is set up in the area from line 200 to line 220. If the number that you type into *Q* is less than one or more than five

there is an emphatic reminder output by line 300 to let you know the computer is paying attention; go again, back to "YOUR TURN" in line 210. When you finally get past these tests, GOSUB 500 is used again to print the results of your pick.

The computer is permitted to play next. The subroutine from line 800 to line 930 is used by your opponent, and it is jumped to from line 270. As can be seen early lines 810 and 820 contain the essence of the programmed strategy. If fewer than six letters remain the computer takes all but the final one (line 900). If more than ten letters are still active, the computer fakes its intelligence with a random guess (lines 840 through 860). When the countdown gets to between ten and six letters remaining the program takes but one at a time (line 920). Sure, you can beat it—but you are a programmer.

Humanizing Points

Two other points need to be made about the mechanics of how this program is controlled. In lines 200, 710, and 880 the symbol *P* (player) has a use. When a game does end either a parting shot (HA HA), or a meek acknowledgment (OOPS) is made—based on whether *P* has a one or a two in it.

REPLAY

Early in the mainline, back up in line 155, the *Q$* was blanked. In line 195, and again in line 260, *Q$* is checked for a *Y* character. After the first game gets under way the only place that accepts input into *Q$* is in line 760. That is where, after a game has ended, the player thumping on the keyboard has a chance to ask for another game. That sequence has to end with a return. (A GOSUB was in force through line 500, to line 650, to line 700.) To start another game the branch reenters the program at line 155 to again blank *Q$*.

And the game goes again; at least until *Z-End*.

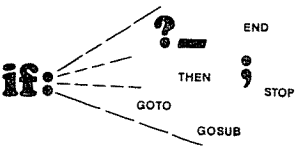
THE PROGRAM

```
10 REM  "Z-END"
20 REM
30 GOSUB 9000
40 PRINT "SKIP THE RULES (Y OR N)";
50 INPUT Q$
60 IF Q$ = "Y" THEN 150
70 PRINT "I'LL PRINT THE ALPHABET."
80 PRINT "YOU'RE FIRST."
90 PRINT "TYPE THE NUMBER OF LETTERS THAT"
```

```
100 PRINT "  I SHOULD OMIT NEXT TIME."
110 PRINT "WE TAKE TURNS AND THE LIMIT"
120 PRINT "  PER TURN IS 5."
130 PRINT "THE ONE THAT GETS THE 'Z' IS"
140 PRINT "  THE LOSER AND THAT'S Z-END."
150 PRINT "GOOD LUCK, CUZ I'M CLEVER..."
155 LET Q$ = " "
160 DATA A,B,C,D,E,F,G,H,I,J,K,L,M
170 DATA N,O,P,Q,R,S,T,U,V,W,X,Y,Z
180 LET Z = 1
185 LET Q = 0
190 GOSUB 500
195 IF Q$ = "Y" THEN 155
200 LET P = 2
205 PRINT
210 PRINT "YOUR TURN";
215 LET Q = 0
220 INPUT Q
230 IF Q < 1 THEN 300
240 IF Q > 5 THEN 300
250 GOSUB 500
260 IF Q$ = "Y" THEN 155
270 GOSUB 800
280 GOTO 190
290 PRINT
300 PRINT "ILLEGAL - 1, 2, 3, 4, OR 5!"
310 GOTO 210
500 LET Z = Z+Q
550 FOR I = 1 TO 27
560 READ Z$
570 IF I = Z THEN 590
580 NEXT I
590 LET J = Z
600 FOR I = J TO 26
610 PRINT Z$;
615 READ Z$
620 NEXT I
630 RESTORE
640 PRINT
650 IF 26-Z = 0 THEN 700
660 RETURN
700 PRINT "Z-END  ";
710 IF P = 1 THEN 780
```

```
720 PRINT "OOPS"
730 PRINT
740 PRINT "DO IT AGAIN (Y OR N)";
750 INPUT Q$
760 IF Q$ = "Y" THEN 660
765 PRINT "GOOD BYE"
770 END
780 PRINT "HA HA!"
790 GOTO 730
800 PRINT
810 IF 26-Z < 6 THEN 900
820 IF 26-Z > 10 THEN 840
830 GOTO 920
840 LET Q = INT(10*RND(1))
850 IF Q < 1 THEN 840
860 IF Q > 5 THEN 840
870 PRINT "MY PICK IS"Q
880 LET P = 1
890 RETURN
900 LET Q = 26-Z
910 GOTO 870
920 LET Q = 1
930 GOTO 870
9000 REM  "RANDOM NUMBER ROUTINE"
9010 LET Z = RND(1)
9020 RETURN
```

Appendix 1
BASIC Dialects: Foreign Conversions



INPUT PROMPTING

During my research for this book I did not find any system that used other than a question mark as a system-provided indicator. This does not mean that your system is abnormal if it does not automatically print a question mark following an input command. What it does mean, however, is that in copying these programs you will have to notice in the player dialog where they should be inserted. The following is typical:

```
PRINT "WANT INSTRUCTIONS (Y OR N)";

INPUT Q$
```

A question is being asked, so when the print is executed it should be followed by a question mark. As a programming convention I expected two services of the system: the character string within the quotation marks would be output, and a question mark would appear automatically immediately following (as a result of the input command). If your system does not supply the question mark simply insert one within the quoted string.

PRINTING

In the above example the trailing semicolon in the print expression has a special significance: the input prompt (?) will appear immediately after the printed question on the same print line. In the

absence of the semicolon the question mark would occur at the start of the next print line. If the syntax of your system digests the semicolon character in some other way one will have to be eliminated. And you will have to achieve the desired results according to your own system's conventions.

CONDITIONAL CONTROL

Normally branching is done with a GOTO statement. The syntax for direct branches seems to be universal enough, but there are deviations for doing conditional branching. The keyword THEN is interpreted as a synonym for GOTO in most BASIC implementations. Some even permit the use of either, in which case the *semantical* results of the following would be the same.

```
IF A = B THEN 100
```

```
IF A = B GOTO 100
```

There is at least one commercial BASIC on the market that does not permit the word THEN. You may have to substitute a GOTO for every instance of THEN in all of these programs. The vast majority of modern BASIC products do work with THEN, however, and I have favored its use for literary reasons.

There is another reason: the BASIC I have used will accommodate virtually any valid construct after the word THEN—not just a line number. For the most part I have used only a line number after THEN to preclude any aggravation for those syntax is restricted. I did enjoy one exception, however, which may require you to make a minor modification. Consider:

```
IF A = B THEN GOSUB 1000
```

The intent, of course, is to do a subroutine jump if the condition is true and to return to the next immediate statement. If the condition is not true program execution falls through to the next statement, bypassing the subroutine jump. There are two other possible ways of doing this, one of which may be inherent to your BASIC.

```
IF A = B GOSUB 1000
```

No problem. Just leave out the word THEN. If neither of these methods works in your case you will have to use the following one.

```
IF A = B THEN nnnn
```

or

IF A = B GOTO *nnnn*

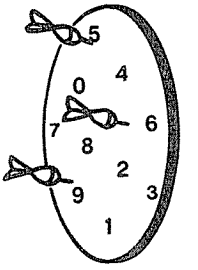
At the address specified (nnnn), then code a GOSUB 1000 and follow it with a GOTO (back to the line number that immediately follows the conditional expression).

END vs STOP

Some BASIC implementations have both of these keywords. Many, during program execution, work in a similar manner to halt the program. That is the logical function. Your BASIC may expect the word END to appear only at the physical end of the program. That is, END should only be used in the last statement of the program.

If your BASIC will permit STOP as a logical terminator simply code in the word STOP where I have used END. And if your system also insists on the END at the end of the program file use it accordingly.

There is one other possibility: you may have to logically execute the END at the end. If so, where I have used END, substitute a GOTO, naming the last statement of your program (where you have added an END statement whose line number is the highest number in the program).



Appendix 2
RANDOMIZE Methods

A tally of the various BASIC missals at my disposal indicates that one of the following should promote the correct results in your machine.

1. Use the one included throughout these programs. The GOSUB 9000 that is coded in the early lines of a program goes to an RND expression that uses a nonzero integer. This is supposed to start the ball rolling, and thereafter each use of RND (with a nonzero integer) will fetch another, unique pseudorandom number.
2. The keyword RANDOMIZE may be substituted for the GOSUB 9000, and lines 9000, 9010, and 9020 may be omitted. If your BASIC likes the word RANDOMIZE it will probably be necessary to remove the parenthetical expressions in all other RND statements as well. Care must be exercised in doing so to the extent that all that is to be removed is the left and right parentheses (and the integer) that is associated with the intended RND(1). Any other parentheses or variables or operator must be preserved.
3. It is possible that you need RANDOMIZE and that you need to omit the parenthetical modifier that follows any RND, and still need to do more. This is usually the case when the system needs to be furnished a *seed*(in some BASIC implementations the seed is generated internally,

often by use of a clock circuit). The following coding will do that, and presumes that you will get there by way of the GOUSB 9000 that already exists—replacing only those lines beyond 9000.

```
9010 PRINT "PLEASE TYPE IN THE DAY OF THE MONTH";
9020 INPUT Q1
9030 PRINT "TYPE IN THE MINUTES PAST THE HOUR";
9040 INPUT Q2
9050 LET Q3 = (Q1+Q2+ABS(Q1-Q2))/2
9060 LET Q4 = (Q1+Q2-ABS(Q1-Q2))/2
9070 IF Q4 > 5 THEN 9090
9080 LET Q4 = Q4+5
9090 FOR I = 1 TO Q3/Q4 STEP Q4/Q3
9100 LET Q5 = RND(Z)
9110 NEXT I
9120 RETURN
```

Appendix 3

Game Matrix

	PLAYERS				GAME TYPE							
	1 PERSON	2 PERSONS	2,3, or 4	COMPUTER	CARDS	DICE	NUMBER GUESSING	LETTER GUESSING	GRID BASED	DATA SORTING	NIM VARIANT
ABSTRACT	•						•				
BANDIT	•						•				
COKES			•				•				
DICE		•				•					
ELEVATE	•						•				
FIVECARD	•			•	•						
GUNNERS		•					•		•		
HOTSHOT	•			•			•		•		
INVERT	•			•						•	
JUSTLUCK	•					•	•				
KNIGHTS	•									•	
LAPIDES		•									•
MATCH			•		•						
NAUGHTS	•			•					•		
O-TELL-O		•							•		
PAR-2			•				•				
QUANTAL		•					•				
ROULETTE	•						•				
STATES		•						•			
TWENTY 1	•			•	•						
ULTRANIM	•	•		•							•
VERBOTEN		•						•			
WAMPUS	•			•			•				
XCHANGE	•	•								•	
YAT-C		•				•					
Z-END	•			•							•

Appendix 4

Truth Table Blanks
8 Bits by 256 Words

WORD NO.	OUTPUTS								REMARKS
	08	07	06	05	04	03	02	01	
0									
1									
2									
3									
4									
5									
6									
7									
8									
9									
10									
11									
12									
13									
14									
15									
16									
17									
18									
19									
20									
21									
22									
23									
24									
25									
26									
27									
28									
29									
30									
31									
32									
33									
34									
35									
36									
37									
38									
39									
40									
41									
42									
43									
44									
45									
46									
47									
48									
49									
50									
51									
52									
53									
54									
55									
56									
57									
58									
59									
60									
61									
62									
63									

WORD NO.	OUTPUTS								REMARKS
	08	07	06	05	04	03	02	01	
64									
65									
66									
67									
68									
69									
70									
71									
72									
73									
74									
75									
76									
77									
78									
79									
80									
81									
82									
83									
84									
85									
86									
87									
88									
89									
90									
91									
92									
93									
94									
95									
96									
97									
98									
99									
100									
101									
102									
103									
104									
105									
106									
107									
108									
109									
110									
111									
112									
113									
114									
115									
116									
117									
118									
119									
120									
121									
122									
123									
124									
125									
126									
127									

continued on page 306

Truth Table Blanks (continued from page 305)

WORD NO.	OUTPUTS								REMARKS
	08	07	06	05	04	03	02	01	
128									
129									
130									
131									
132									
133									
134									
135									
136									
137									
138									
139									
140									
141									
142									
143									
144									
145									
146									
147									
148									
149									
150									
151									
152									
153									
154									
155									
156									
157									
158									
159									
160									
161									
162									
163									
164									
165									
166									
167									
168									
169									
170									
171									
172									
173									
174									
175									
176									
177									
178									
179									
180									
181									
246									
183									
184									
185									
186									
187									
188									
189									
190									
191									

WORD NO.	OUTPUTS								REMARKS
	08	07	06	05	04	03	02	01	
192									
193									
194									
195									
196									
197									
198									
199									
200									
201									
202									
203									
204									
205									
206									
207									
208									
209									
210									
211									
212									
213									
214									
215									
216									
217									
218									
219									
220									
221									
222									
223									
224									
225									
226									
227									
228									
229									
230									
231									
232									
233									
234									
235									
236									
237									
238									
239									
240									
241									
242									
243									
244									
245									
246									
247									
248									
249									
250									
251									
252									
253									
254									
255									

Index

A

Abstract 15
- building the program 17
- design approach 16
- how the program works 19
- program 22
- programming problem 16

B

Bandit, defining the problem 27
- design strategy 27
- program 31
- the internals of 28

Battleship 76

C

Card code generation 57
Chuck-a-Luck 115, 118
Cokes, how the program plays 35
- program 36

Column 77
Command module 82

D

Dice, general-purpose technique 40
- how works 40
- problem definition 38
- program 44
- the architecture of 40

Dimension 40
Draw 65

E

Elevate, how the program works 48
- program 51

Errors 49

F

Fini-key 26
Fivecard, architecture 63
- clear table-H 67
- clear table R 67
- design 55
- frequency counts 67
- hand analyzer 66
- how mainline works 64
- how works 56
- table A 57
- table A1 57
- table A2 62
- table B 57
- table B1 57
- table B2 62
- table D 56
- table H 58
- table R 60
- the program 70
- winner picker 69

Flush 56, 68
- straight 68

Full house 68

G

Gunners, building 77
- deploying the tanks 81
- four clues 76
- internals 79
- program 83

H

Hand 58
- analyzer 58

High card 68
Hot shot 87
- making work 92
- program 95
- programmed strategy 89
- putting together 90

Human factors 48

I

Indubitable deck 146
Invalid shot 92
Invert, check for a winner 109
- computer's pick 108
- display 2 tables 106
- player input 107
- player's numbers 108
- program 110

program overview 103
what's inside 105

J

Justluck, program 120
architecture 116
listing 118

K

Knights from within 128
layout 127
program 133

L

Lapides, check for win 140
logic 138
player input 140
print 140
program 141
Logical grid 88

M

Match from the inside 149
Match from the outside 145
Match, program 152
Maximum range 48
Meat of the matter 66
Micros 38
Multiples 67

N

Naughts & Crosses,
design approach 156
housekeeping 162
mainline 162
modules, mapping & more 160
tables 158
the program 167

O

O-tell-O, basic logic principles 176
from the inside 179
picture of the program 178
play 172
problem definition 174
program 183

P

Par-2, design considerations 191
program 195
programming 193
Picture-name print statements 27

Q

Quantal, design & logic 200
program 202

R

Rank 56
Roulette, playing 206
program 211
programming 207
Row 77

S

Shuffle 148
States internally 219
States, program 223
States template 218
Straights 67
Structured programming 63

T

Tank 253
Twenty1, design audit 229
program 235
putting together 230
the deck 228
Two-shot 93

U

Ultranim, program 247
rest of 243
strategies in nim 240
the computer's choice 241

V

Verboten program 252, 256
Verboten's coding 254

W

Wampus, describing the
mechanics 262
describing the
program 260, 262
program 265

X

Xchange, end 277
game 270
print grids 276
program 277
program organization 271
salient points 272
task orientation 271

Y

Yat-C, coding 286
program 289
program template 284

Z

Z-End, from the top 294
program 296
rules 295